COMS W4735: Project Write-Up
Jimmy Hughes

1 Introduction

This paper presents the methods and results of my computer program that integrates with a webcam allowing the user to construct 3D objects. Similar to Tilt Brush, this program allows the user to paint in 3D space by using common items as a brush. The user will need to find an object with a handle on one end and a spherical object on the other (e.g., lollipop, percussion mallet, wooden dowel with a ball attached to the end). The user then measures the radius of the sphere and completes a one-time calibration step in which the object is measured in image space. After calibration the program tracks the spherical object in 3D space allowing the user to construct 3D objects by "painting" in 3D space.

There are some limits to this program. As mentioned, above the app must be calibrated before use. This will entail measuring the object and holding it at a fixed distance from the camera. Out of the box the program is calibrated to track a sphere with a specific shade of blue. Thus, the user will have to adjust the threshold boundaries for whichever object they decide to use. Useful tools for that calibration are included in the *utils.py* file. The program assumes the camera remains at a fixed location pointing in a fixed direction at all times. The app will work in varying light intensities (more below) but there should be enough light for the camera to distinguish between colors. The app was tested on my laptop but it should work on any laptop with a webcam and Linux operating system.

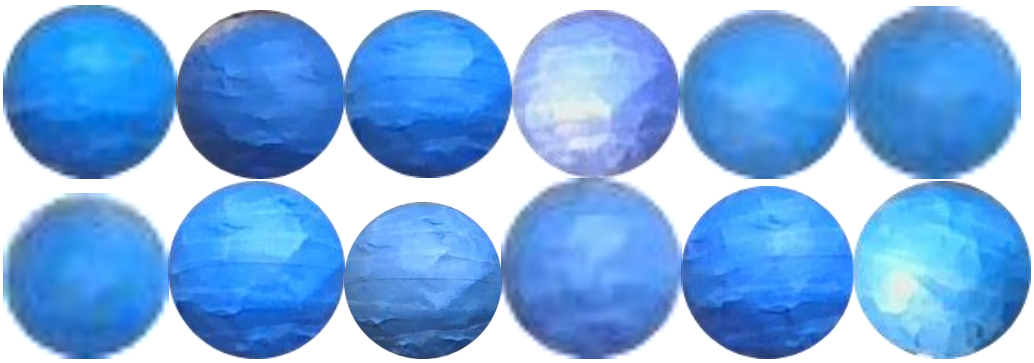
2 Method Description

In this program I made use of the following python libraries: OpenCV, numpy, matplotlib, and pykalman. The app's pipeline can be broken down into the following processes: *obtaining 2D coordinate*, *calibration, estimating 3D coordinate, object tracking* and *3D object construction*.

2.1 Obtaining 2D Coordinate

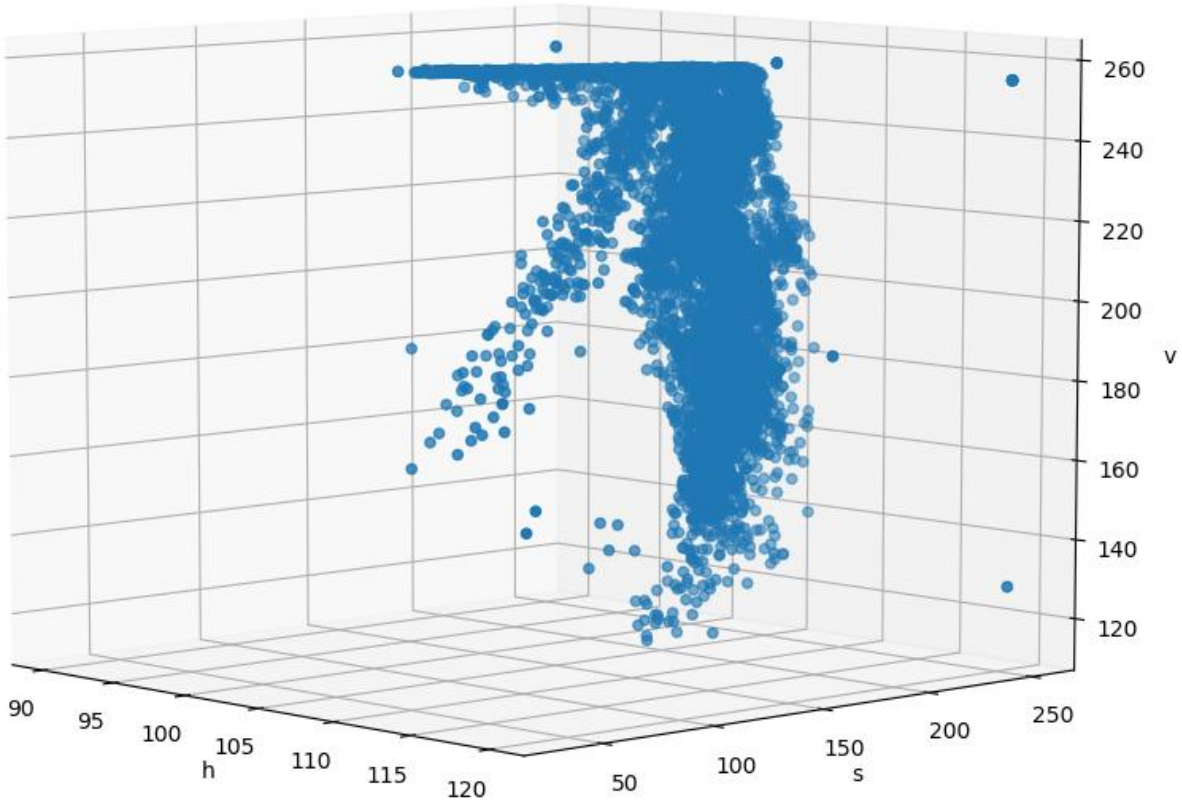
The first aspect of this program I handled was detecting the image space coordinate of the spherical object. The reason I chose a sphere as the object of interest is because it will always project as a circle in image space regardless of its angle relative to the camera. To begin I scaled the image down to 337x600 resolution. This allowed for faster processing time while still preserving enough fidelity in estimating the sphere's location. I then I applied a Gaussian blur and converted the image from RGB to HSV. I wanted to apply a simple color segmentation algorithm that used threshold boundaries to segment the image into sphere and non-sphere pixels but as I tested my RGB threshold values in different lighting environments it became impossible to find a set of boundaries that would work in all settings. This is because the RGB color space mixes color related information and light related information into all three values. In HSV, however, the H component only contains information about the absolute color while the S and V values store information about the lighting1. In order to find threshold boundary values I collected a batch of images under varying illumination conditions (time of day, varying lighting, different rooms) and I cropped my sphere out of each image.

[1]Color spaces in OpenCV, Vikas Gupta, 07 May 2017



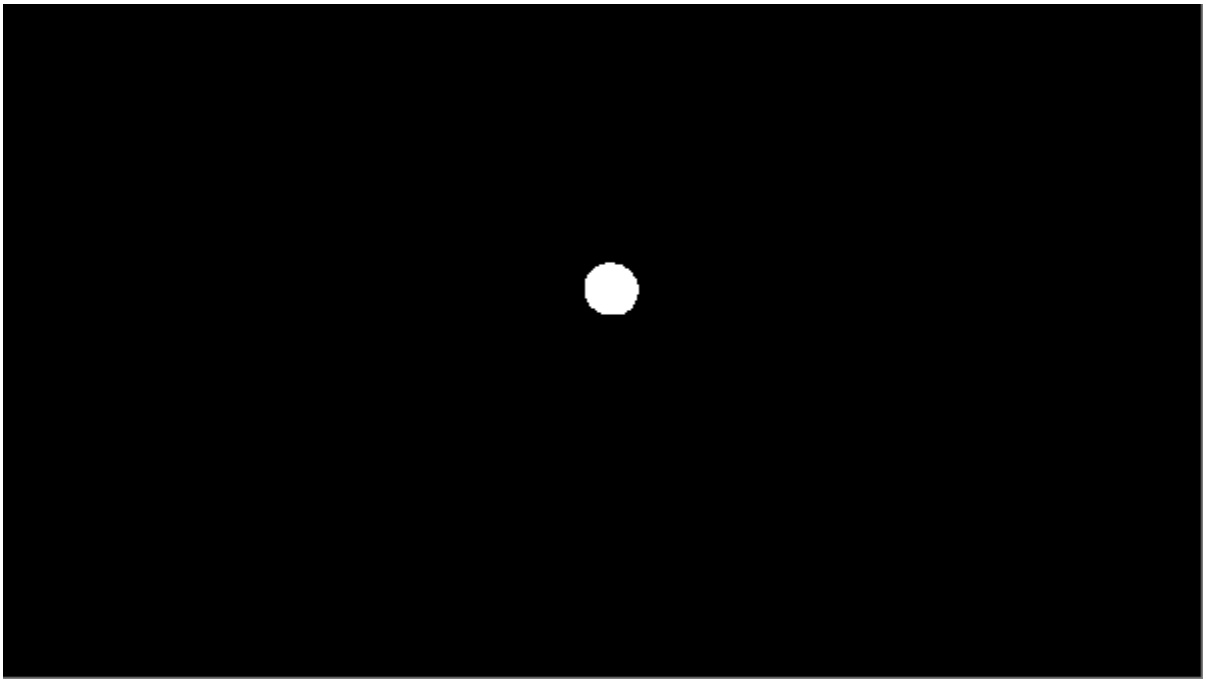
A sample of the images I collected for threshold values.

Then I created a three-dimensional scatter plot from the HSV value of each pixel in the cropped images. The code for generating this plot can be found in the *utils.py* file.



Three-dimensional scatter plot of HSV values of the collected sphere samples.

This allowed me to gauge which HSV volume the sphere pixels are concentrated in. I was able to select boundaries the work best in many lighting environments. After applying a threshold to the image using the new found boundaries I applied OpenCV functions to dilate and erode the image to remove any remaining small blobs. After all of the above processes the image clearly segments the sphere pixels from the non-sphere pixels.



Webcam image with sphere of interest after resizing, applying Gaussian blur, threshold, erosion, and dilation.

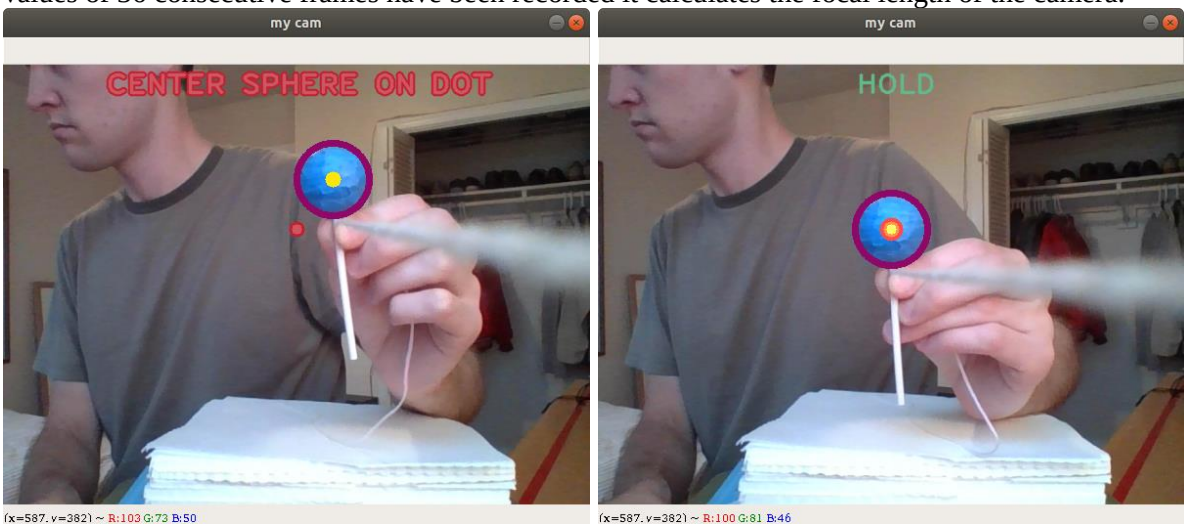
I then made use of OpenCV's contour implementation to find the blob with the largest area in the segmented image. I then used OpenCV's minimum enclosing circle function to find the circular boundary of the sphere. This gave me the radius and center of the sphere. I then took the center of that circle as the object's image space coordinate and took its radius to calculate its 3D coordinate (see section 2.3).

2.2 Calibration

In the calibration step the program uses information about the physical environment along with information from the image space to determine properties about the camera which will be useful for estimating the object's 3D coordinate. To start, the user must measure the radius of the spherical object. I found the best way to do this was to tighten a C-clamp on to the sphere then take it off and measure the distance. In my case the radius was $11/16^{th}$ of an inch. This value should be stored in the TRUE_RADIUS variable on line 150 of *final.py*. Next the user must decide on a distance from the camera to hold the sphere. I chose 24 inches and I stored the value on line 151 as CALIBRATION_DISTANCE. Note, the user can measure these values using any length unit but they must be consistent (i.e., if the radius is measured in mm then the calibration distance must also be measured in mm). After entering these values the program will ask the user to center the sphere in the frame at the provided distance and hold it for 36 frames. To help assure the sphere is at the correct distance from the camera I taped a piece of string to the laptop and marked where 24 inches was.

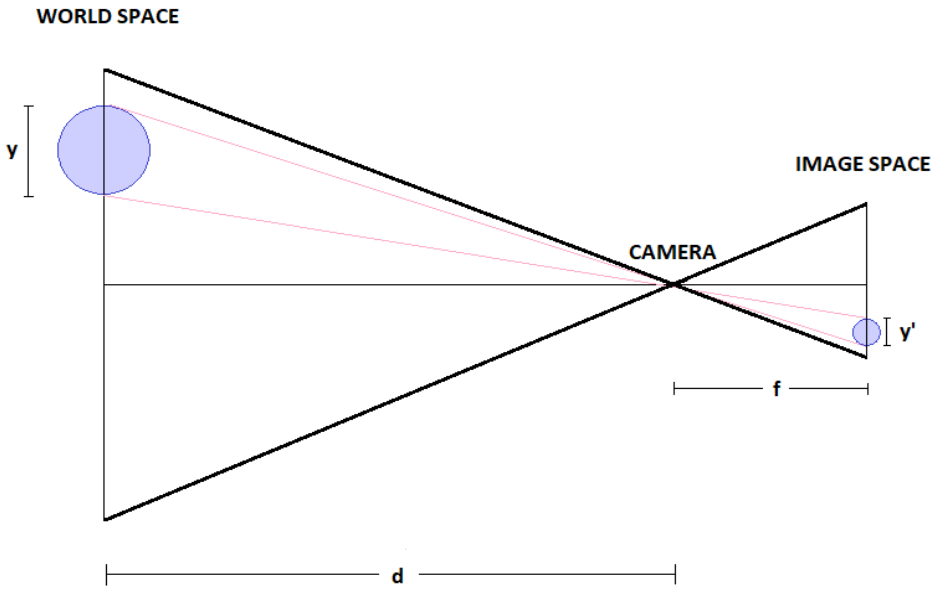
This is how I calibrated the program. The string is measured to be a fixed length from the laptop camera.

As the user holds the sphere in place the program will record the radius of the circle in image space at each frame. If the object moves away from center the calibration process will restart. After the values of 36 consecutive frames have been recorded it calculates the focal length of the camera.



Screenshots of the program during the calibration process.

The recorded radii are averaged giving us what we call the observed radius. Given the observed radius along with the two values provided by the user we can measure the camera's focal length. This can be visualized below. The working distance, *d*, and object height (sphere radius), *y*, were provided by the user and the observed radius, *y'*, is measure by the program.

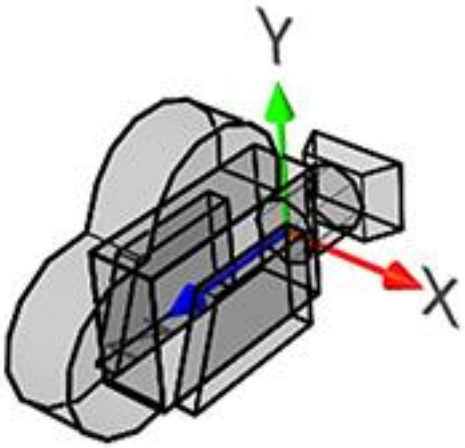


Relationship between observed and true measurements.

Thus, we can find the focal length, *f*, by using triangle similarity: $f = \frac{d \times y\prime}{y}$. That concludes the calibration step. After calculating the focal length we can calculate the distance from the camera to the sphere given its perceived radius. This is just another application of triangle similarity: $d = \frac{f \times y}{y\prime}$. After finding the working distance of the sphere we can calculate the world coordinate of the object using trigonometry.

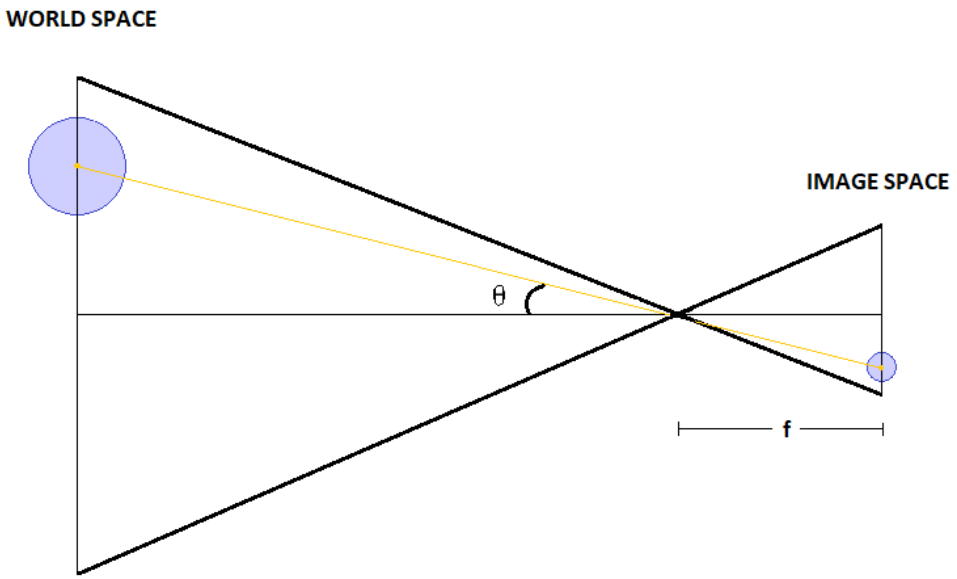
2.3 2D to 3D Mapping

After estimating the depth of the object based on the observed radius the program estimates the coordinate of the sphere center in the world space. Before discussing the details of that calculation I want to define the world coordinate system. The units depend on how the user calibrates the program, for example, since I calibrated using inches the output will be in inches. The program takes the origin to be the center of the camera. The negative z-axis is pointing away from the camera, positive x-axis pointing to the right and positive y-axis pointing towards the top of the image. Similarly, we define the image space coordinate system to have an origin at the center of the image and x and y axes pointing in the same direction as the world space coordinate system.



Visualization of the camera coordinate system.

After given the depth and image space coordinate of the center of the sphere we can calculate the 3D position of the center by applying trigonometry. First, we find the angle between the x-value and the z-axis. We will call this value θ. Since we know the focal length and the image space coordinate we can use $\theta = arctan(\frac{x}{f})$.



Relationship between image space coordinate, *x,* focal length, *f,* and angle ϑ.

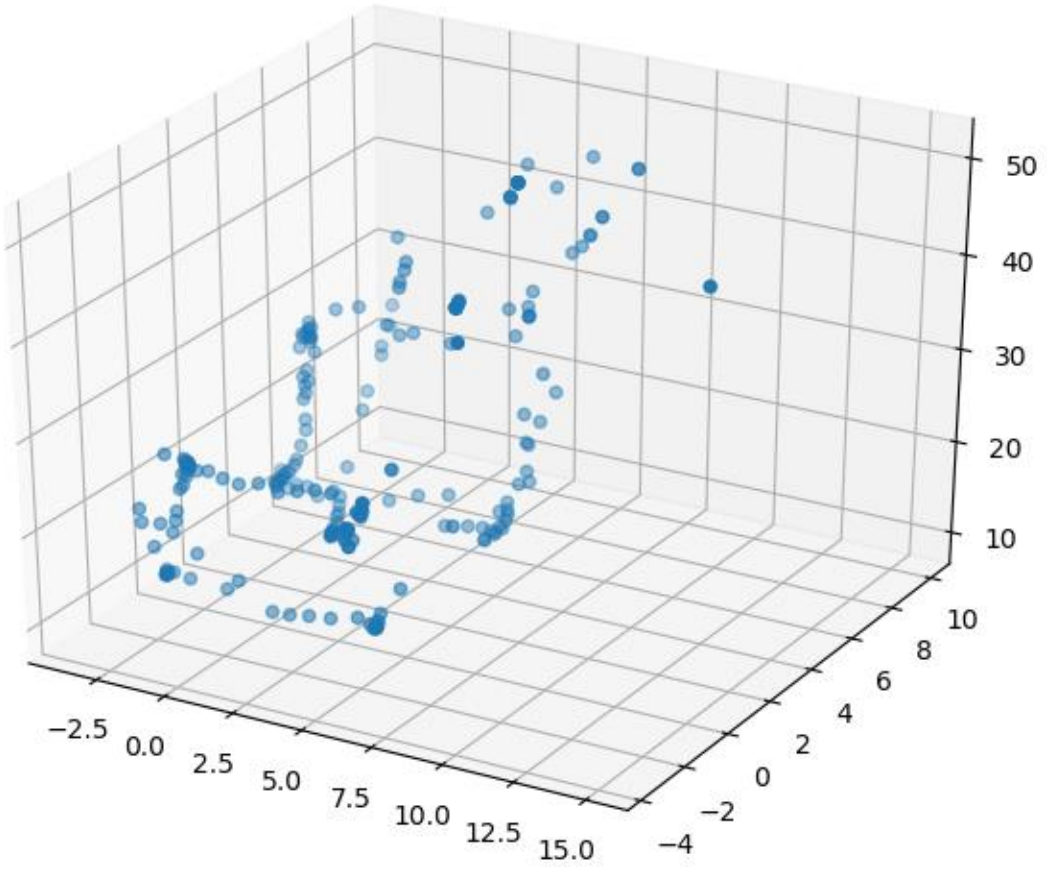
Similarly, for the angle between the y-value and the z-axis we can use $\varphi = arctan(\frac{y}{f})$. Then from the depth we have $x = d * \tan\theta$ and $y = d * \tan\varphi$.

2.4 Tracking Problems

In the process of constructing this program I ran into several difficulties, mainly during the process of tracking the sphere. I have documented some of those issues and solutions below to shed light on my process and explain why I made some of the choices I made in this program.

2.4.1 Color of Sphere

Originally, my sphere was bright pink and semi-translucent. I decided, however, to wrap my sphere in blue painter's tape because during the segmentation step parts of my skin would occasionally be marked as sphere pixels. This was normally not an issue but it would occasionally cause the program to start tracking my skin or increase in radius when the sphere would pass in front of parts of my body. This made some of the renderings noisy such as the plot below which was supposed to be a cube.



Scatter plot of cube sketch with noise.

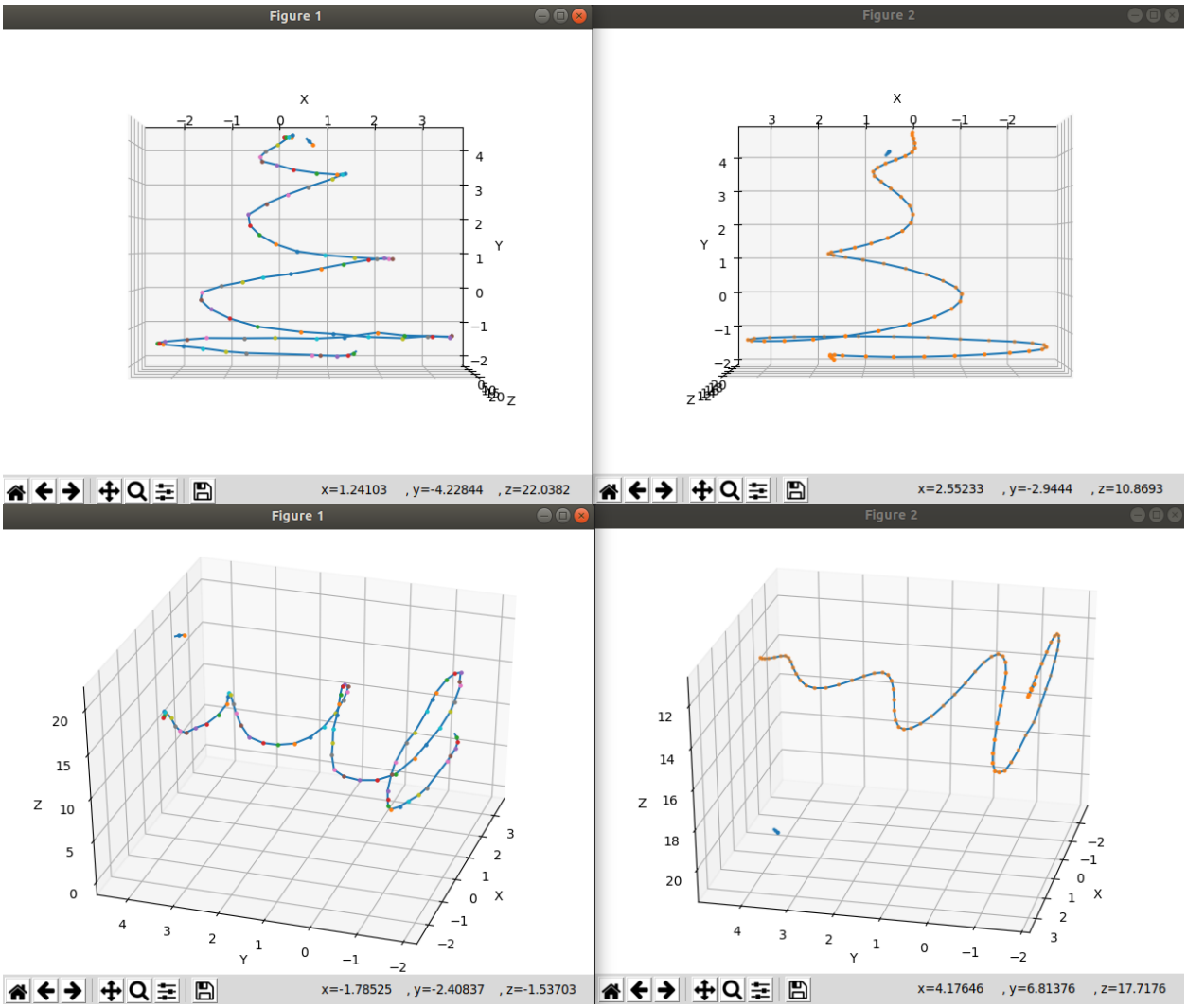
2.4.2 CSRT & KCF

To smoothen noise during tracking I also experimented with several tracking algorithms. This was easy to play with since OpenCV has several built in methods for tracking. I tried OpenCV's implementation of the CSRT and KCF algorithms. Both, worked fine in tracking the general movement of the sphere but the size and position of the estimated bounding box around the sphere was very inaccurate. I decided to stick with the detection/tracking methods listed above because they were much more accurate for size and location and they worked fast enough to track in real-time.

2.4.3 Kalman Filter

I did however still occasionally get noisy results during tracking using the above methods and I wanted to find a way of making the final result smoother. To smoothen my final set of points I used Kalman filtering which is an algorithm that takes a set of observed data that's assumed to contain statistical noise and attempts to estimate its error covariance and minimize it with a new set of data[2].

After the estimating the 3D points during tracking the program applies Kalman filtration to each set of points.



Left two images: raw data from 3D estimation. Right: data after applying Kalman filter.

2.5 Displaying Results

During tracking, as the program gathers 3D points for each frame representing the center of the sphere, it plots each of the points in real-time in a three dimensional scatter plot using matplotlib. Furthermore, a line segment is drawn between consecutive points in real-time. I also added a feature where the user can draw multiple unconnected lines on the same plot by lifting the space bar between

[2]Implementation of Kalman Filter, Mohamed Laaraiedh

strokes. After the user is done drawing and they exit the canvas mode by pressing escape and each of the line are processed through a Kalman filter and displayed on a new window. The final 3D object can be inspected closer at different angles in the window.

3 Results

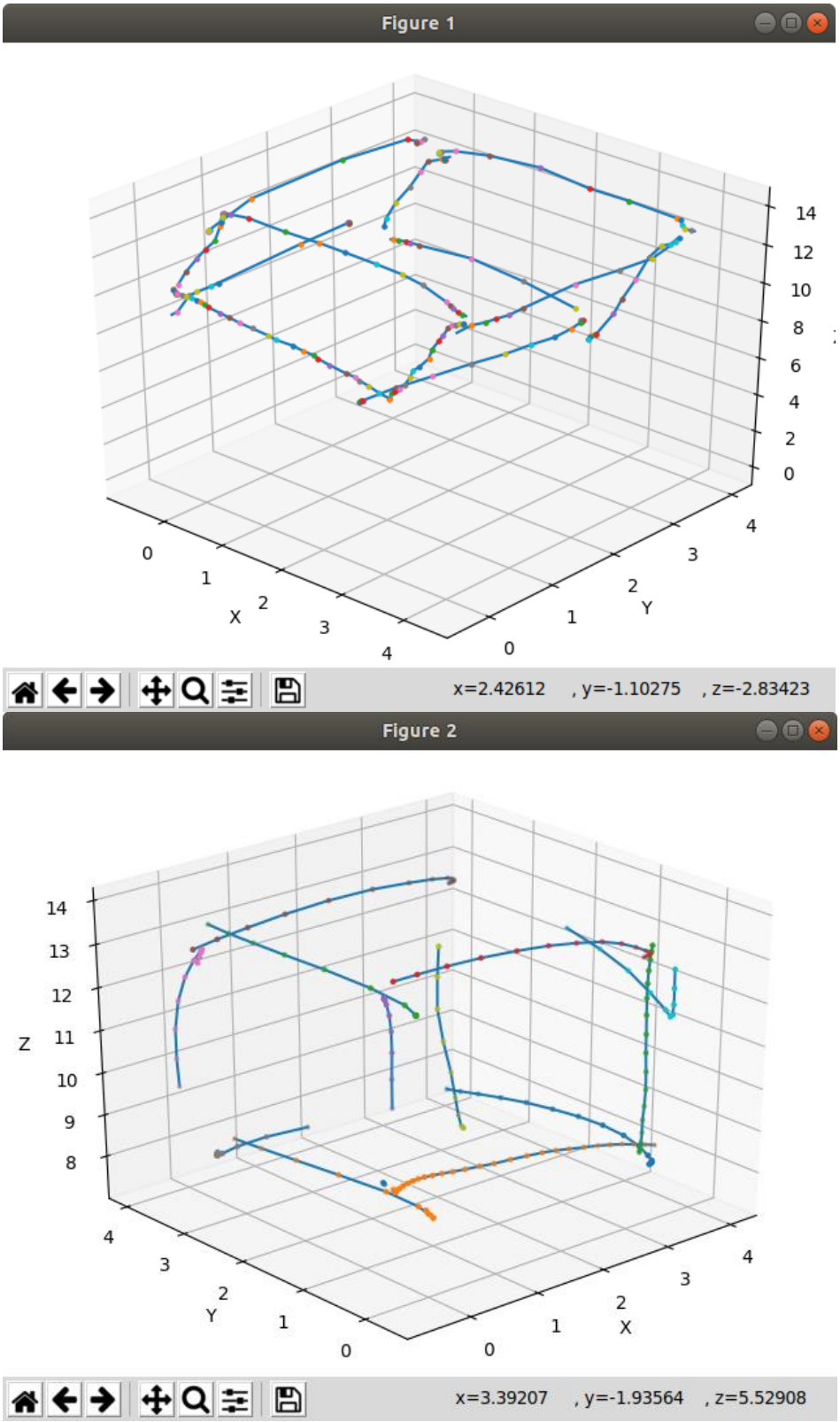
The app is able to successfully detect and track the spherical object and estimate where it is in 3D space. I evaluated the detection and tracking by viewing the circular boundary around the sphere in the calibration mode and canvas mode. To evaluate how well the program estimates the sphere's 3D location I performed two types of tests, qualitative and quantitative. In the qualitative test I constructed known 3D dimensional objects and viewed resulting 3D object. The results of these tests are subjective because it is up to the user how much the generated object looks like what they were attempting to create. With that said, I believe the program performed strongly in this field.

In the second type of test I performed I attempted to make it more objective. I tested the program estimations against known locations along a grid and compared values.

3.1 Drawing 3D Objects

3.1.1 Cube

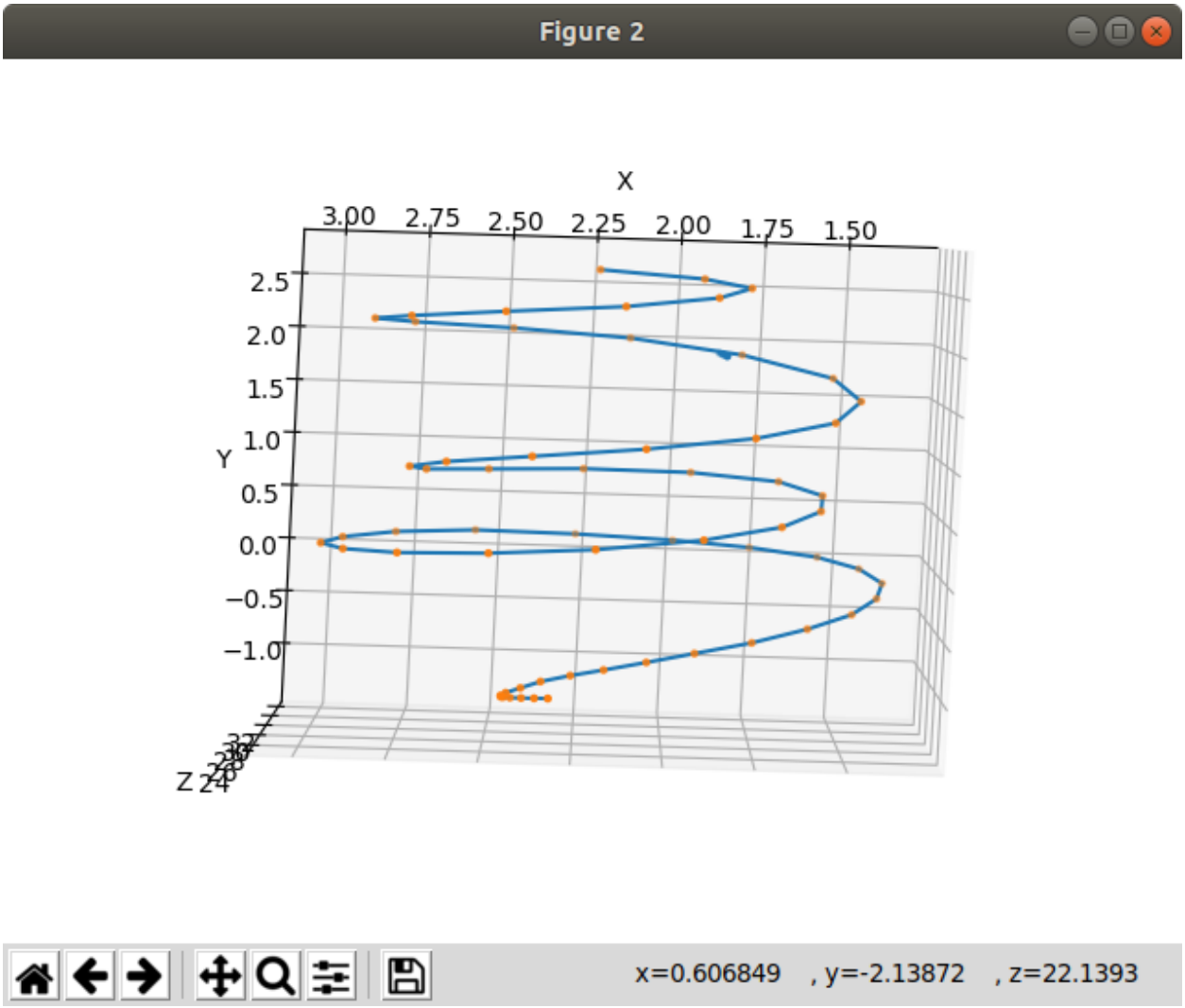
In this test I attempted to draw a cube. The first image below is the raw input and the second image is the data after applying Kalman filters. I think this was fairly successful. It became difficult to track where the edges started and stopped when I was drawing so some of the edges don't connect correctly but that is more of a design issue rather than an algorithm issue.



Cube before and after Kalman filter.

3.1.2 Helix

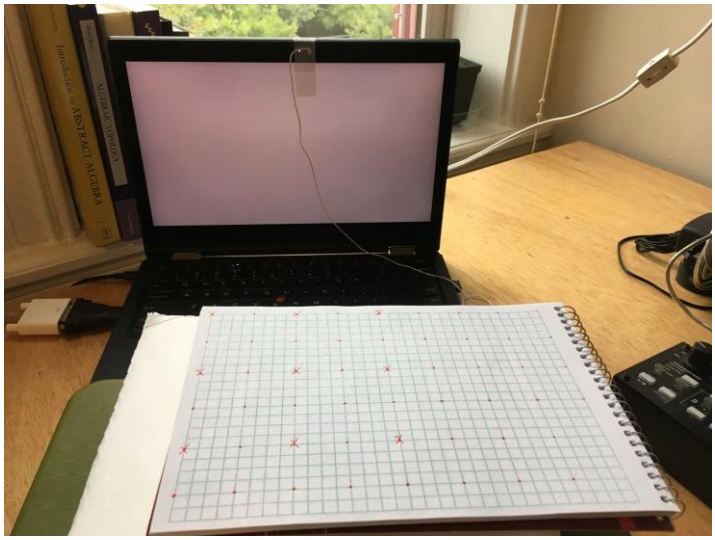
The second shape I attempted to draw was a helix. Again, it was slightly difficult to draw a perfect helix since I had to guess where I was drawing before but I am still satisfied with the results.



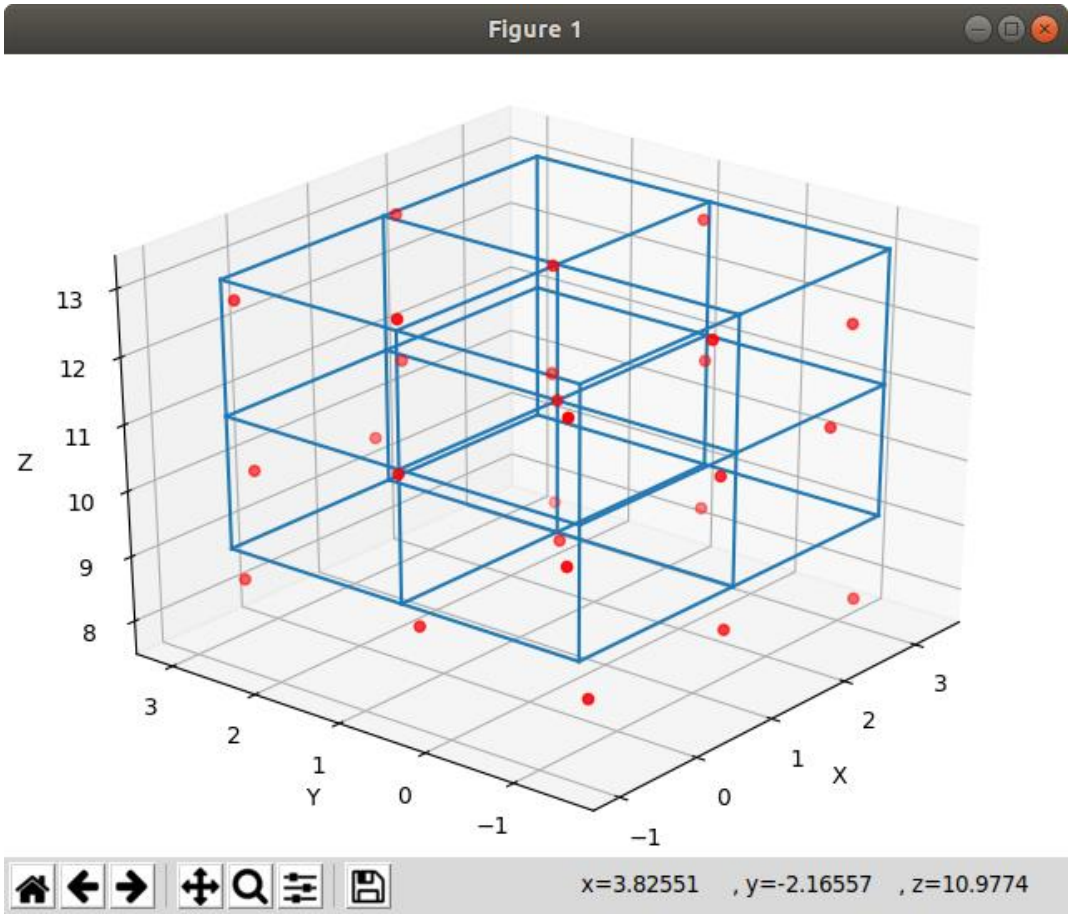
Helix after applying a Kalman filter.

3.2 Grid

To test against known values I created a three by three grid spaced at two inch increments. I then placed that in the center of the webcam such that the grid ran along the x-axis and z-axis of the world coordinate system. After everything was in place I placed the sphere at the nine points on the grid and recorded their positions in the program. After recording those points I raised the graph paper two inches by propping the paper on top of a stack of books. I continued to measure the nine points then raised it two inches again and measured them one last time. After recording all twenty seven sample points I plotted them in a scatter plot along with the expected values.



Graph paper positioned in front of webcam for testing.



The recorded points are in red while the expected values are in blue.

I am pretty satisfied with these results. To measure the quality of these estimates I calculated the distance from each point to its neighboring points along each axis. After that I averaged those distances and found the distances along the x-axis averaged to be about 1.96 inches, y-axis averaged to about 1.90

inches and z-axis averaged to 1.99 inches. This is to be expected since the points were spaced at intervals of two inches. The results of the recorded coordinates against the expected points were less impressive. I found an average distance of 2.77 inches. I think this discrepancy can be credited to the grid not being entirely centered in the webcam because it got shifted as I stacked books underneath it. Also, the webcam was not perfectly perpendicular to the surface so the z-axis might not have lined up correctly with the real grid. For those reasons, I think the relative metric is a better gauge of performance than the absolute metric. In the former the program did very well. The code used to calculate these results can be found in *utils.py*.

4 Conclusion

This paper presented methods integrating with a webcam to allow the user to construct 3D objects, allowing the user to paint in 3D space. Given an object with a handle on one end and a spherical object on the other I have presented a technique for estimating detection and tracking in 3D space. I then showed a way to smoothen the results and construct/display a 3D object. We saw that while the user interface was less than ideal in some cases, the mechanics of the program worked quite well and were fairly accurate. In future work I would like to take this a step further by displaying results in an augmented reality setting so the user can see what they are painting displaying in 3D space around them.

5 References

1) Color spaces in OpenCV, Vikas Gupta, 07 May 2017 *https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/*
2) Implementation of Kalman Filter, Mohamed Laaraiedh https://arxiv.org/pdf/1204.0375.pdf