ADEV Image Correlation
End of Quarter Report

Dr. McCormack
ME 471-02

Dustin George
Drew Maione
Timothy McDaniel

20 February 2017

System Overview

GitHub Repository: https://github.com/mcdanitd/ADEV-Image-Correlation/

For our capstone design project, we have been tasked with designing a system capable of correlating the images of each of three camera sensors for an automatic strawberry harvester made by ADEV Automation. The robot's sensor package contains two webcams in the RGB color space and a time-of-flight (ToF) infrared depth camera that will provide us distance information.

Correlation in the computer vision context refers to the use of the offset between two cameras to gain information about a three dimensional space. If the offset between the cameras is known, then the difference in the images they provide can be used to find information about the space that they can see. This is similar to how the human eye develops and uses depth perception, but we have the added advantage of a depth camera for added precision.

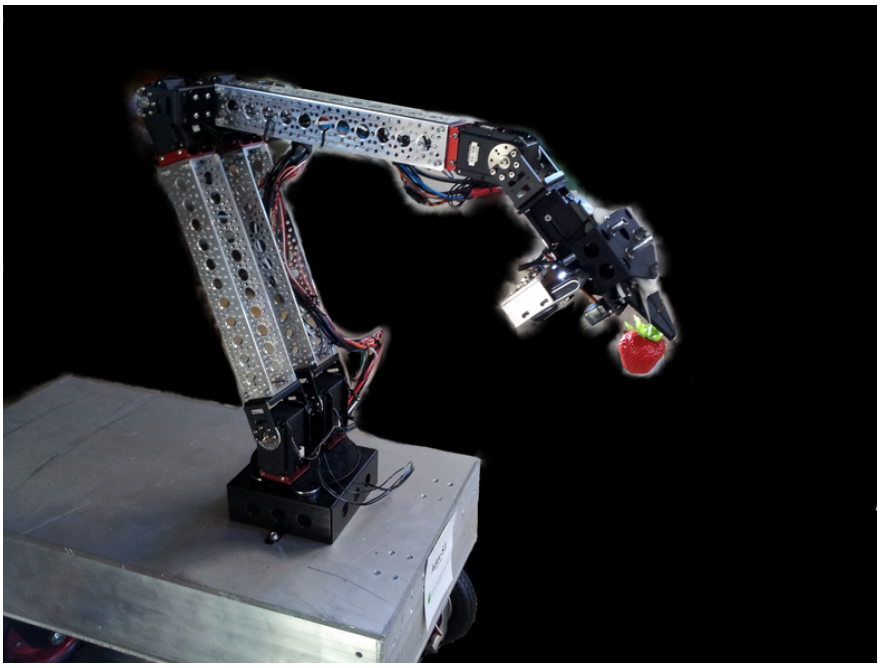

Figure 1. *A close up of ADEV's strawberry picking prototype, the S-1, picking strawberries like a champ.*

To date, we have built both a software and a hardware component to our system. The main component of our project and the task we have been asked to accomplish is an image correlation software package. This package uses mostly Python 2.7.5 with OpenCV and NumPy packages to handle our data input and array calculations with some C++ to support our depth camera's API.

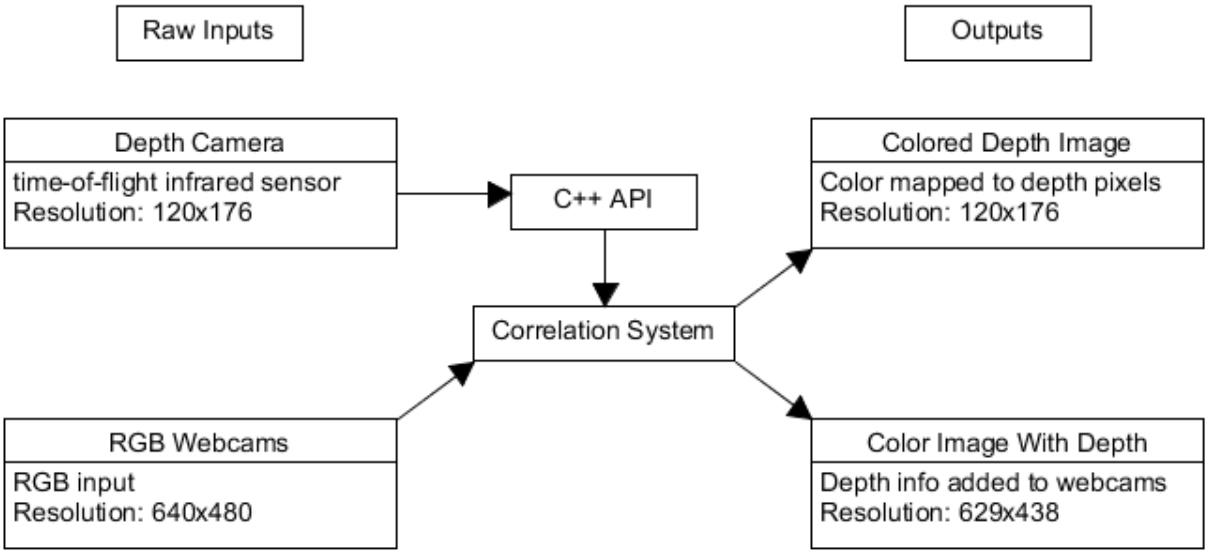

Figure 2. Flowchart overview of our system's inputs and outputs. Note that removing distortion from our webcams' image resulted in some lost pixels [1].

The correlation package primarily uses geometric relations to map the image from the webcams onto the image from the depth cam, and vice versa. This allows a user or strawberry identification program to request the distance to any point in the webcam's view.

The camera mount is designed to hold both of the webcams and the depth-of-field camera, with their lenses in near-perfect alignment. This will allow the correlation package to achieve the highest accuracy. It must also be able to interface with the robot arm.

Correlation Process

Presentation of design

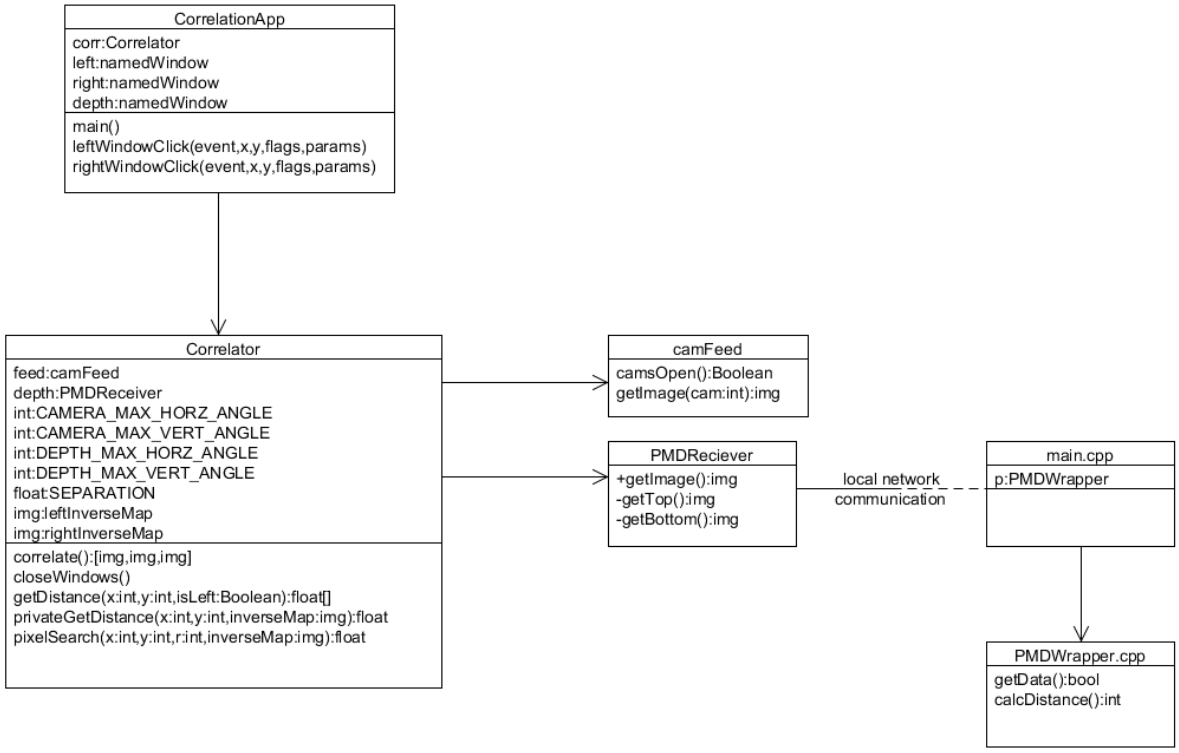

Figure 3. *Full UML diagram for the ADEV Image Correlation project*

Our system reads input from three separate cameras and prints distance information to the console when the user clicks on a window displaying a webcam's video feed. Above is the full UML diagram for our project. Programs we have written that are not actively used by the main system will not be discussed in this section, but can be found in the Appendix. The following sections will present each class's software requirements individually.

Depth Camera Feed (PMDReciever.py, main.cpp, PMDWrapper.cpp)

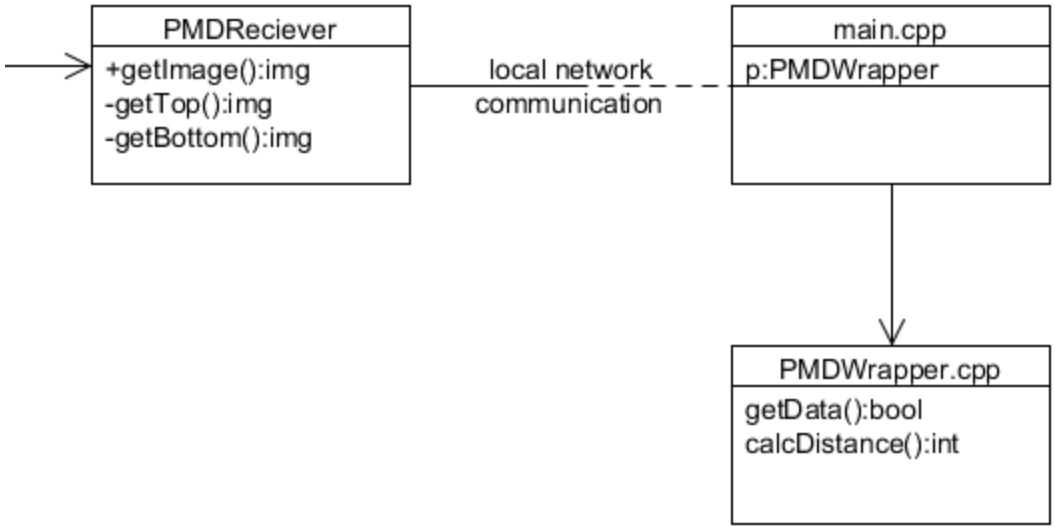

***Figure 4**. Relevant parts of the UML diagram for reading from the PMD depth camera*

Reads depth sensor using PMD API.

The C++ script PMDWrapper.cpp is used by main.cpp to read the raw depth information from the depth camera using the API provided by PMD, the company who manufactured our depth camera, for using their sensor. This section of the project must be in C++ because we have not been provided a Python API for this camera.

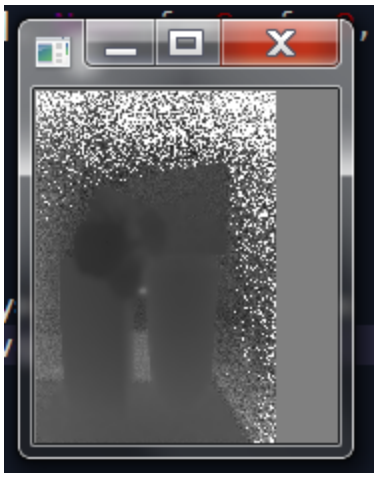

Figure 5. An example of a depth camera image; dark areas are closer to the camera, lighter areas are further away, and static areas are false pixels too far away for the sensor to see.

Splits data array into two arrays.

The depth camera's output uses too much bandwidth to be sent as a single image. Each frame of the depth camera's feed is split into two halves that are sent separately.

Adds identifier to each array.

After being split into two halves, a marker is added to the corner of each image to identify which image represents the top and bottom halves.

Sends arrays as data packets on a local network port.

Each of the two image arrays is sent as a packet over a local network port to the main body of the project.

Receives PMD feed from PmdUdpDumper.cpp in the form of two data packets.

PMDReciever.py takes in two depth camera packets at a time from the local network port.

Correctly orders the two data packets into one image.

After receiving both halves of an image, PMDReciever.py recombines them into a single image. The markers that were written into the image are used to identify which half is the top image and which is the bottom image.

Outputs depth array

When prompted by Correlator.py, returns the ordered depth image.

Webcam Feed (camFeed.py)

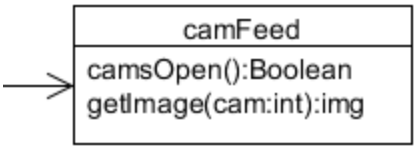

Figure 6. *camFeed.py as shown in our UML*

Reads webcam feed from a single camera.

An instance of this class will read input from the webcam it is pointed at. We use OpenCV to open the webcam and to read data from it.

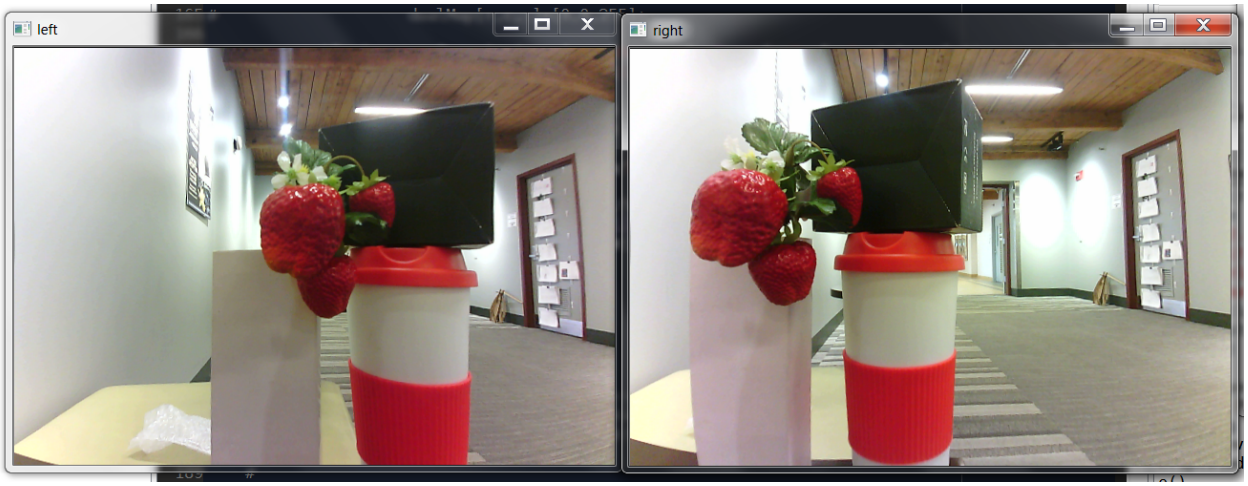

Figure 7. An example of the stereo webcam input that our system receives. The offset in distance between our cameras' mounting points gives us slightly different images from right to left.

Calibrates video feed to remove radial barrel distortion.

The lenses in the RGB webcams add a "fisheye" effect to the video feed. Using functions built into NumPy and OpenCV's Python packages, we are able to calibrate each frame of our video feed so that our data accurately represents the real world. These functions resulted in a few lost pixels, but the team decided that the lost pixels represented a negligible loss in information. The resolution of our webcam image dropped from 640x480 to 629x438. With distortion removed, we can correctly use a cartesian coordinate system in our correlation algorithm.

Outputs webcam data.

When prompted by Correlator.py, returns an undistorted RGB image.

Correlation Algorithm (Correlator.py)

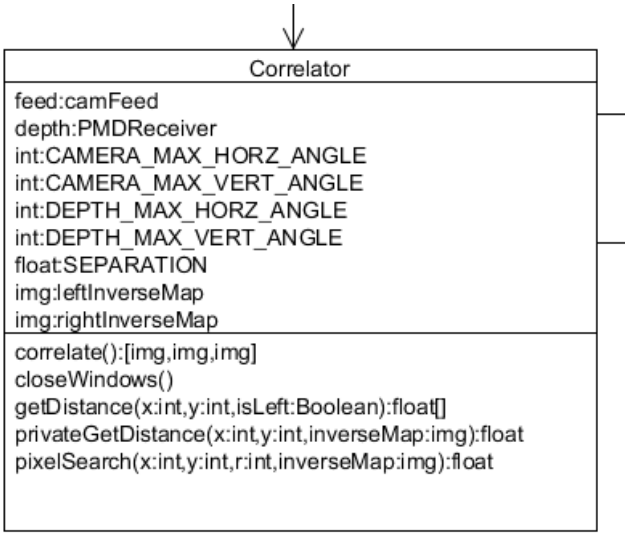

Figure 8. *Correlator.py as shown in our UML*

Import depth camera information and RGB video feeds.

This class gathers the information that other scripts are reading from the sensors.

Correlates the RGB webcams' pixels to each depth camera pixel.

Using the known distance between the cameras, the field of view provided by each camera, and the raw pixel information from each image, we use simple trigonometry to correlate each pixel of the depth camera to a color from the left or right webcam. This gives us a left correlated image, where the RGB information from the left webcam has been matched to pixels on the depth camera and a right correlated image using the right webcam. The left and right correlated images are then combined to form a single, correlated image. Another matrix the size of the webcam image is also made for each webcam which maps distance values from the depth camera to pixels in the RGB image.

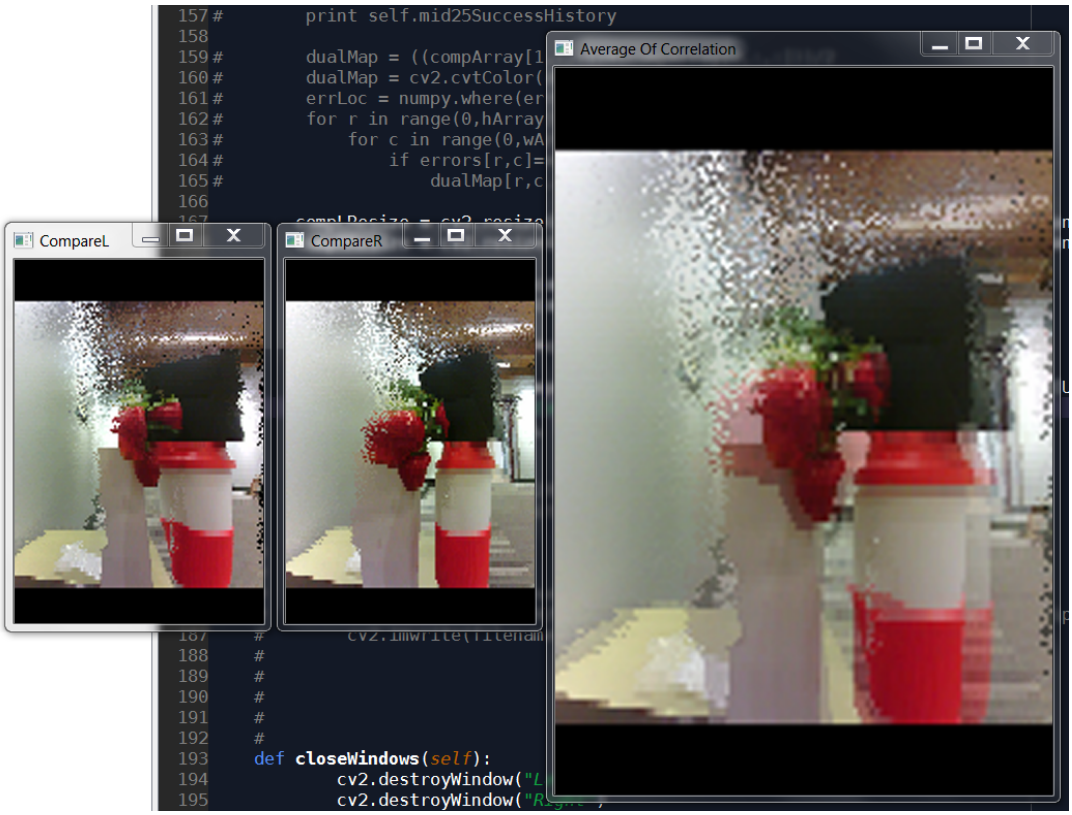

Figure 9. The depth correlated images in our project. Left: Each pixel of the depth camera has had a pixel from the left-hand webcam mapped to it. Center: Each pixel of the depth camera has had a pixel from the right-hand webcam mapped to it. Right: The average of the other two images produces our final correlated depth camera image. These images were taken with our 3D printed ABS plastic mount, and it's inability to hold our cameras in a consistent position from test to test is the most likely cause of the double vision effect seen here.

It is worth noting that the difference in resolutions between the webcams and depth sensor means that the two images generated here are not one-to-one. Each pixel on the depth camera has a color from the webcams assigned to it, but many pixels on the webcam do not have depth pixel to be mapped to. Conversely, only a few pixels in the webcam image will have a depth value assigned to them, as seen in figure 8.

0	.12	0	0	0	.13	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
.07	0	0	0	0	0	.15	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	.05	0	0	0	0	.09

Figure 10. Example of section of color image with correlated depth replacing color pixel data.

Method returns x, y, and z displacement from cameras to selected object in webcam's view

Given a pixel's coordinates in the webcam and which webcam was clicked on, recursively finds the nearest pixel in that webcam that has been correlated to a depth camera pixel. Based on that depth pixel's distance value and direction vector, a method will return the changes in x, y and z.

Allow for accuracy testing.

Here we allow our system to run tests of how accurate our system is. This will be elaborated on in the Performance status section further in the report.

Display video feeds of left and right correlated images and final correlated image.

Using OpenCV, display the video feeds of the left correlated image, the right correlated image, and the final correlated image that combines the two in named windows to the user's screen.

Top Level Script and GUI (CorrelationApp.py)

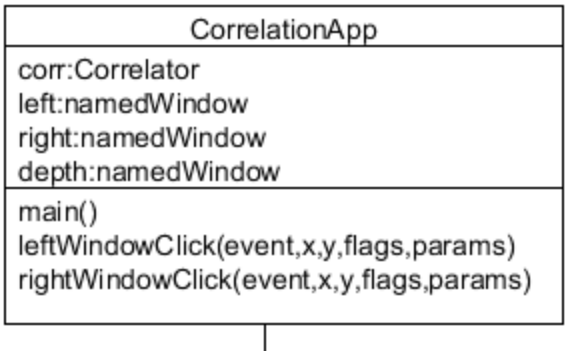

Figure 11. *CorrelationApp.py as shown in our UML*

Display video feeds of left and right webcams and depth camera.

Shows raw data input from our sensors to the user's screen in named windows.

Set up click listener for both left and right webcam feed

This script creates a click listener for the windows showing the left and right webcam feeds. When the user clicks in these windows, the listener will pass the coordinates of the click and which window was clicked on into Correlator's getDistance method, then print to the python console the relative distance from the depth camera's lens to the object that the user clicked on.

```
dx: -0.00772579912341
dy: 0.141464035194
dz: 0.0365966224875
StraightLineDistance: 0.150088888407
```

Figure 12. Console output from getDistance.

Performance Status

Self-Check Accuracy - As an initial test of our system's accuracy, the team wrote code that compared the color value of correlated pixels to check if they were looking at the same object. If our correlation system thought that two pixels should be seeing the same point in space and their values were within a threshold range, then we counted that pixel as accurate, otherwise we considered it incorrect. Heavily dependent on the exact positioning of the cameras, this accuracy check gave widely varied values dependent on exactly how the cameras were placed in the mount and suffered from having a poor mounting method. Because of this, results from this test vary between 30 and 85 percent.

This method of accuracy testing has the benefit of being quick and easy to implement, however it has the disadvantage of not always giving a perfectly representative value for accuracy. For

example, when looking at a scene with a lot of one color, this accuracy test will naturally return better accuracy values since a lot of the pixels are the same color. This problem could be avoided by more carefully selecting the test scene, but even then, there can still be issues with two webcam pixels being matched with each other, but not to the correct depth pixel. With these issues in mind, we have started exploring better ways to test how well the system works.

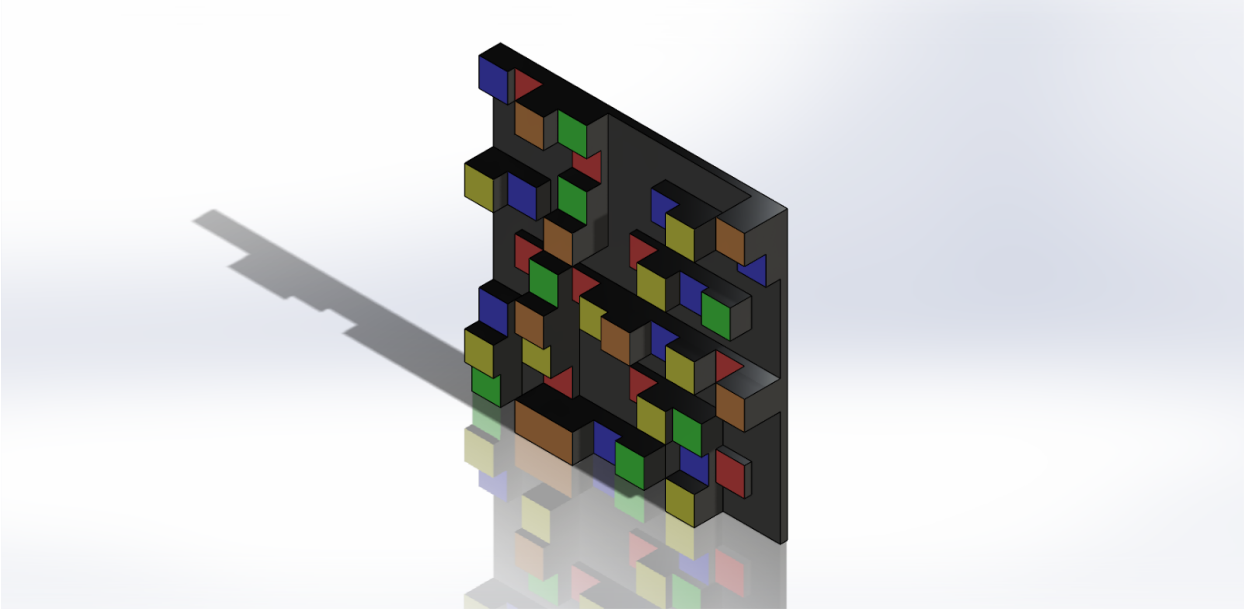

Figure 13. *Concept of test scene that would be used for grid accuracy*

Grid Accuracy - When first starting the project, the team developed an accuracy test involving the reconstruction of a 3D grid with varicolored surfaces. This accuracy test would theoretically give a perfect understanding of how well the correlation algorithm works, since the rebuilt grid would require both distance and color. There are two issues with using this testing system. First, the scene and testing code would not be very easy to construct and write, which, while not necessarily insurmountable, would take time away from doing other, more productive, things. Secondly, the depth camera has been documented to return different distances based on the color of a surface, which would present a problem with the multitude of colors. While the team can easily map reported distance to actual distance for each color, the multitude of colors would present an issue which would require several iterations of guessing and checking, which would slow down the program. While this accuracy test can give the best absolute accuracy, it has some issues with being implemented, and absolute accuracy is not necessarily the most relevant metric for testing the performance of the system because our system is looking for a specific object in a field and not trying to accurately find every object in our cameras' field of view.

Functional accuracy - While the accuracy of the correlation system obtained from the grid test and self check test are good metrics, the ability of the system to accurately return the location of an identified strawberry is the only thing our clients are truly concerned with. As such, a better way of testing our system's performance is to test how well it can locate strawberries relative to the

mount's location. This accuracy test has the benefit of being fairly easy to implement, as there is little additional code that needs to be written and only a basic mount to hold the strawberry needs to be built or even just found. Additionally, we will be able to apply the previously mentioned distance remapping since it will only be concerned with one color. For these reasons, at this point in the project, this is the accuracy test that we will be developing and using.

Presentation of Plan

In the next quarter, we would like to finish this system and add a few features. Once we have an accurate mount for stable testing, we'll be able to add in more software requirements to increase our accuracy. When we know that our mount is accurate to a decently high tolerance, we will be able to take another look at the location of our cameras' apertures. Without detailed schematics from the manufacturers, we have had to take informed estimates about where in the camera each sensor is. So far we have not been able to evaluate if our estimates were correct or not because the inaccuracy created by our mount has outweighed the error that could be caused by this. If the sensor in our webcams was half a centimeter further back than we expected, then we can adjust for that in software.

When we have a mount that is stable and milled accurately, we will be able to perform accurate checks on self, grid, and functional accuracy. Right now, our mount does not reliably hold the cameras in a way that ensures that the cameras are all parallel and on the same level. The correlation algorithm depends on assuming these things to be true. Once we can be sure that physical sources of error have been removed, we can begin to get accuracy values for the correlation and begin to tweak the algorithm to reduce error.

A feature we would like to add is an edge detector for our depth information. A filter designed to detect edges could be used to define discrete objects in our field of view and a function to combine edgels into solid edges. We would want to implement this on the depth camera's view instead of the webcam's color input, because otherwise an object with two different colors would be seen as two objects.

Mount

Presentation of Design

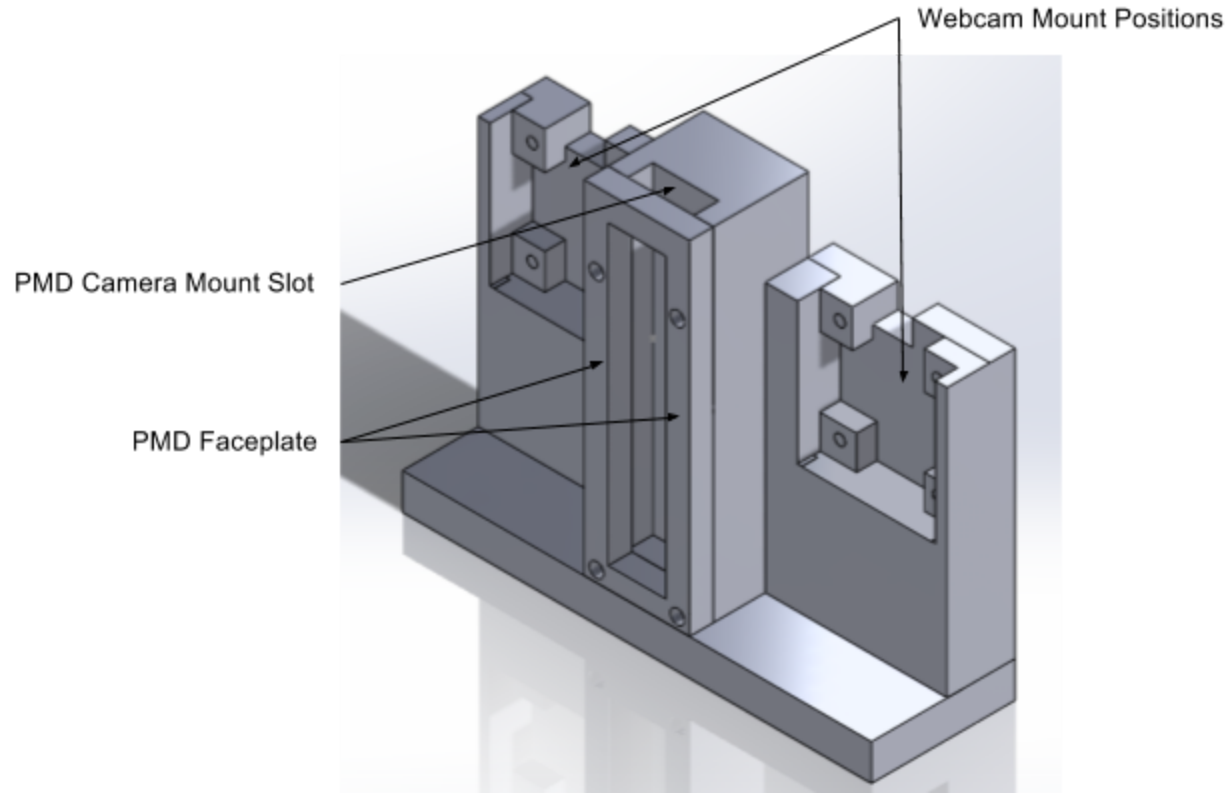

Figure 14. *Labeled assembly of aluminum mount, including the faceplate to secure the depth camera.*

The current design of the mount is rendered in SolidWorks below and has been modified based on problems encountered in mounting the webcams using the previous iteration of the design.

Performance Status

The 3D printed prototype of the mount works as a baseline test of our software but is inadequate for practical application and actual “object finding” within the camera’s field of vision. The problem primarily stems from the fact that the current mount is printed using an older version of the mount design that does not take into account the instability in the webcams. The version of the mount that we are using is pictured below:

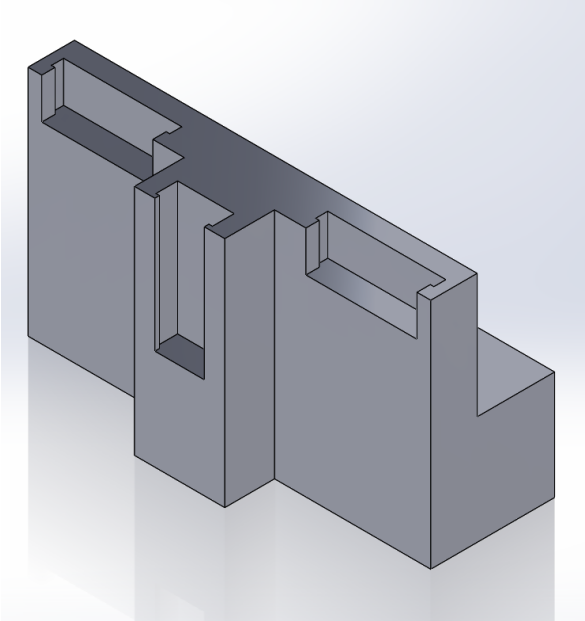

Figure 15. *Original design for the 3D printed mount.*

The problem with the webcams arises because this mount is designed to fit the webcams as though they were simple three dimensional boxes with flat sides. While this is satisfactory for the sides of the webcam circuit board, unfortunately, the back of the circuit board is has electronic circuitry that protrudes from the board. As a result, even though the dimensions and tolerances fit relatively well, the webcam has a tendency to move about quite a bit while sitting in the mount. This poor fit results in lower accuracy when performing the grid accuracy test described above. In order to better secure the webcams, the mount was redesigned to use the four through-holes in the corners of the webcam circuit board. This way, the cameras will sit level and will not move about during operation. See through-holes below:

Figure 16. *Through holes on webcam.*

The mount will be made from material that excels at dissipating heat.

With the depth camera easily capable of temperatures of at least 116 °F, the team has fabricated a new mount prototype out of aluminum 2024 due to the fact that temperatures higher than 80 °F can be dangerous to electronic devices. Aluminum 2024 has a thermal conductivity of 121 W/mK and was selected due to availability [2].

Principle fabrication was completed on the aluminum mount; milled out of 2024 aluminum. Already, we can see that, on the next iteration, it would be wise to mark a datum on the original block of aluminum so we maintain a fix on where the edges of the cameras ought to be and where the lenses line up. For ideal alignment, these measurements need to be made to the tenth of a millimeter. The completed mount can be see below.

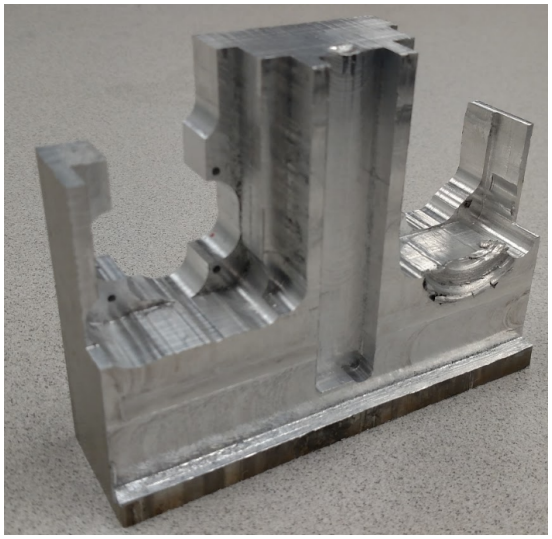

Figure 17. *Completed aluminum prototype mount.*

Milled parts have a tolerance of 10 thou to 1.5 thou [3]. Even if our final mount is on the high end of these tolerances, we expect this to be sufficient improvement over our 3D printed ABS version. From experience, student made and maintained 3D printers can be expected to operate between 20 and 30 thou of tolerance. With the aluminum mount taking advantage of the through holes in the webcams to mount them, it also gives them a fixed position on the mount. With the cameras being held in the same position every time we test, any error can be compensated with in our software.

Presentation of Plan

Next quarter, the team will focus on refining the mount fabrication process and making the aluminum mount more usable. While the first fabrication of the aluminum mount possessed all of the correct alignment dimensions, it was fabricated entirely by hand using a mill with little computer assistance. As a result, the areas hollowed out for the webcams are imprecisely cut (they don't fit the webcams properly). This problem can be remedied by instead using a waterjet

cutter to cut away the material to form the webcam mount positions. After this process is complete, we can simply mill down vertically to get the appropriate material height measurements without damaging horizontal alignment of the cameras.

We also intend to use the initial aluminum prototype as a test model to determine if we can bring the weight of the prototype down. Given that the mount is going to be mounted on a robot arm on the ADEV S-1, ADEV has decided that the mount must weigh no more than 0.5 lbs. Currently, the “completed” prototype weighs around 1.45lbs. However, there is plenty of excess material that can be removed and the team is confident that this part can reach ADEV’s weight goal.

References

[1] “Reference Design Brief CamBoard picoS 71.19k.” Internet: http://pmdtec.com/html/pdf/PMD_RD_Brief_CB_pico_71.19k_V0103.pdf, [20 Feburary 2017].

[2] “Aluminum 2024-T4; 2024-T351.” Internet: http://www.matweb.com/search/datasheet_print.aspx?matguid=67d8cd7c00a04ba29b618484f7ff.7524, [20 February 2017]

[3] “Typical Tolerances of Manufacturing Processes.” Internet: http://www2.mae.ufl.edu/designlab/Lab%20Assignments/EML2322L-Tolerances.pdf, [20 February 2017].

Appendix

For the online turn in, please refer to the files that came in this zipped folder.

1. UML Diagram
2. Coordinate System
3. Needs, Metrics, and Software Requirements
5. PMDWrapper.h
6. PMDWrapper.cpp
9. Main.py
13. PMDReciever.py
14. Correlator.py
20. camFeed.py
22. CorrelationApp.py
23. SOLIDWORKS Drawing for 3D Printed Mount
24. SOLIDWORKS Drawing for Aluminum Mount