ADEV Image Correlation
End of Quarter Report

Dr. McCormack
ME 472-01

Dustin George
Drew Maione
Timothy McDaniel

19 May 2017

System Overview

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

For our capstone design project, we were tasked to design and build 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 distance between the cameras is known, then the difference in the images they provide can be used to find the displacement to an object of interest. This is similar to how the human eye develops and uses depth perception, but we have the advantage of a depth camera for added precision.

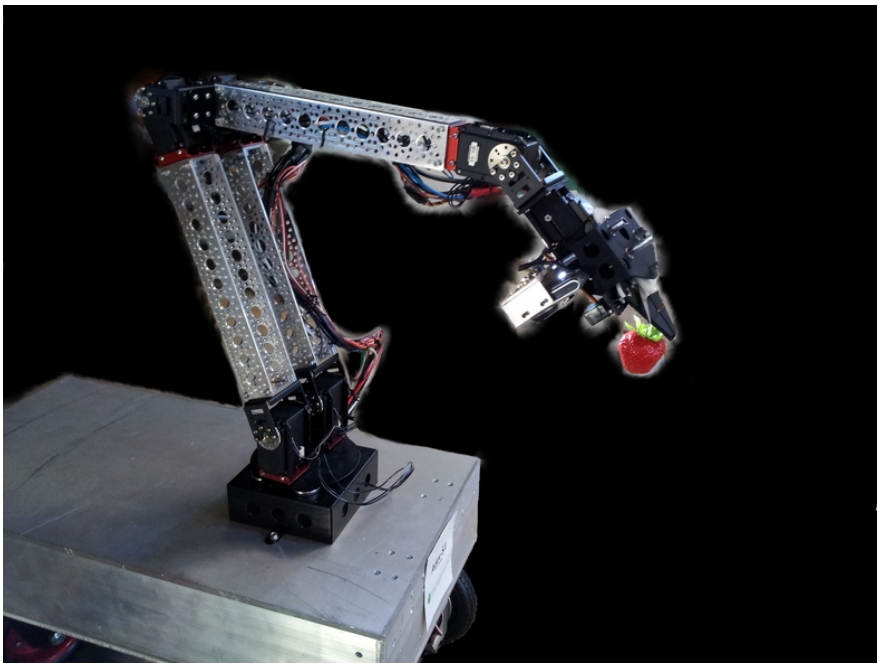

Figure 1. *A close up of ADEV's strawberry picking prototype, the S-1. Their current system does not have an imaging system and requires external guidance.*

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 and we have expanded on that to include additional features that ADEV has requested.

This package uses Python 2.7.5 with OpenCV and NumPy packages to handle our data input and array calculations with C++ to support our depth camera's API.

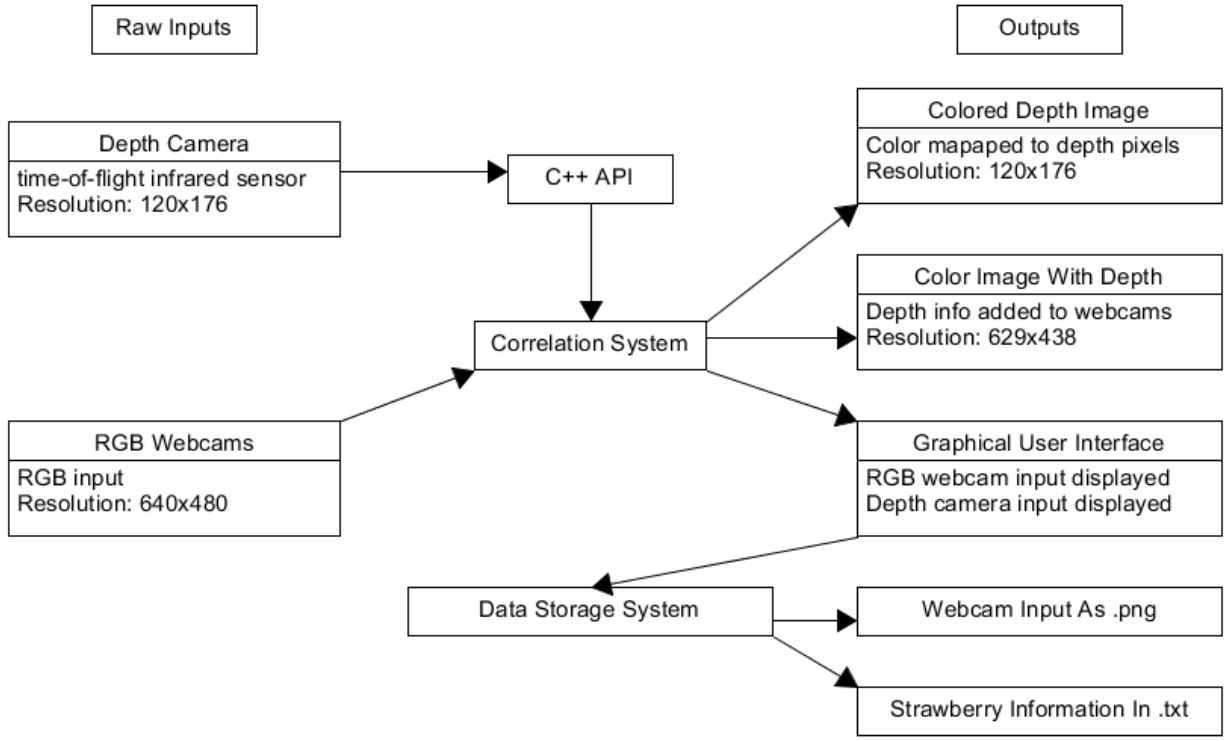

Figure 2. Flowchart overview of our system's inputs and outputs. Removing distortion from our webcams' removed the edges of each image, decreasing the overall resolution [1].

The correlation package uses geometric relations to map the image from the webcams onto the image from the depth cam and vice versa. This allows a strawberry identification program or a user to request the distance to an object displayed in either of the webcams.

In addition to the correlation package, the team developed an image recognition package that is capable of counting the number of strawberries currently in view and saving images and other data in time-stamped .png and .txt files. See the software requirements for DataStorage.py below for details.

The camera mount is designed to hold both of the webcams and the depth-of-field camera such that the lenses are horizontally aligned with an equal spacing between them. This alignment simplifies the geometric relations used to calculate correlation. It must also be able to attach to the robot arm. For details, see the Performance Status section of the Mount report below.

Correlation Process

Presentation of design

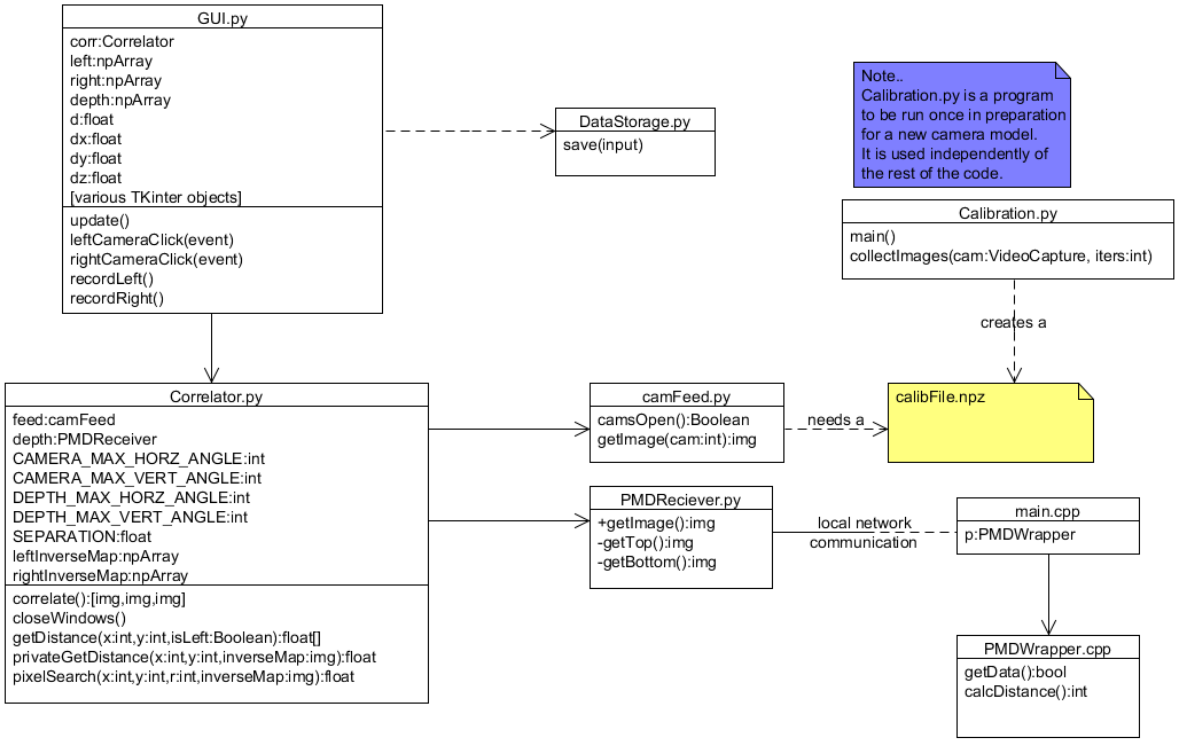

Figure 3. *Full UML diagram for the ADEV Image Correlation project*

This system reads input from the webcams and depth sensor and displays them in a GUI. When the user clicks on a point in either of the webcam displays, the distance to the object in is displayed on the window in terms of x, y, z, and total displacement. At the user's request, the system will count the number of visible strawberries in a webcam's view and save the relevant information in a .txt document and the current camera frame in a .png image file. A screencap of the GUI may be found in Figure 13.

The full UML diagram for the system can be seen in Figure 3. Programs we have written that are not actively used by the correlation system will not be discussed in this section, but can be found in the UML above and in the Appendix. The following sections will present each class's individual software requirements.

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

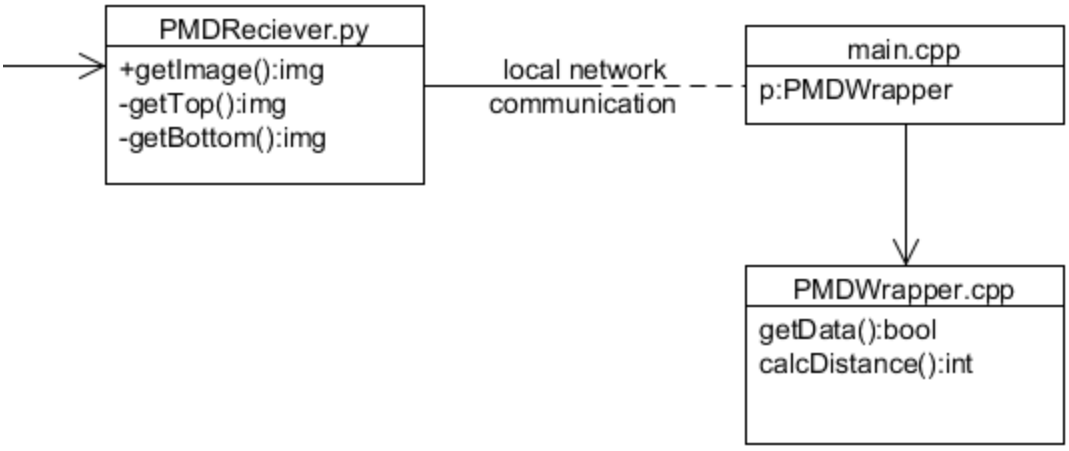

***Figure 4**. Relevant parts of the UML diagram for reading from the PMD depth camera*

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 depth camera's manufacturer - for using their sensor. This section of the project must be in C++ because PMD does not provide a Python API.

The depth camera has a limited range. In the raw data depth camera data from Figure 5, out of range pixels, or false pixels, are seen as static static. Within main.cpp, we use PMD's API to identify false pixels. These pixels are flagged so our correlation system will ignore them.

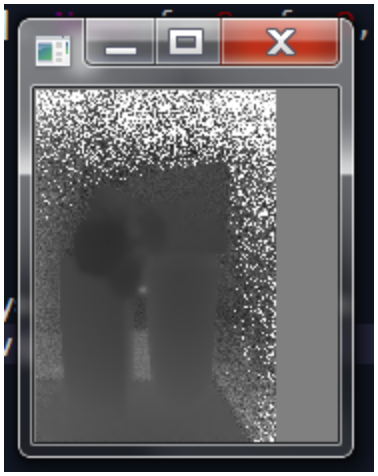

Figure 5. An example of a raw depth camera image; dark areas are closer to the camera, lighter areas are further away, and snowy areas are false pixels too far away for the sensor to see.

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 vertically 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 properly reconstituted depth image.

Webcam Feed (camFeed.py)

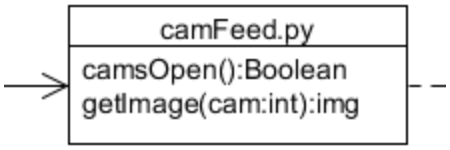

Figure 6. *camFeed.py as shown in our UML*

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.

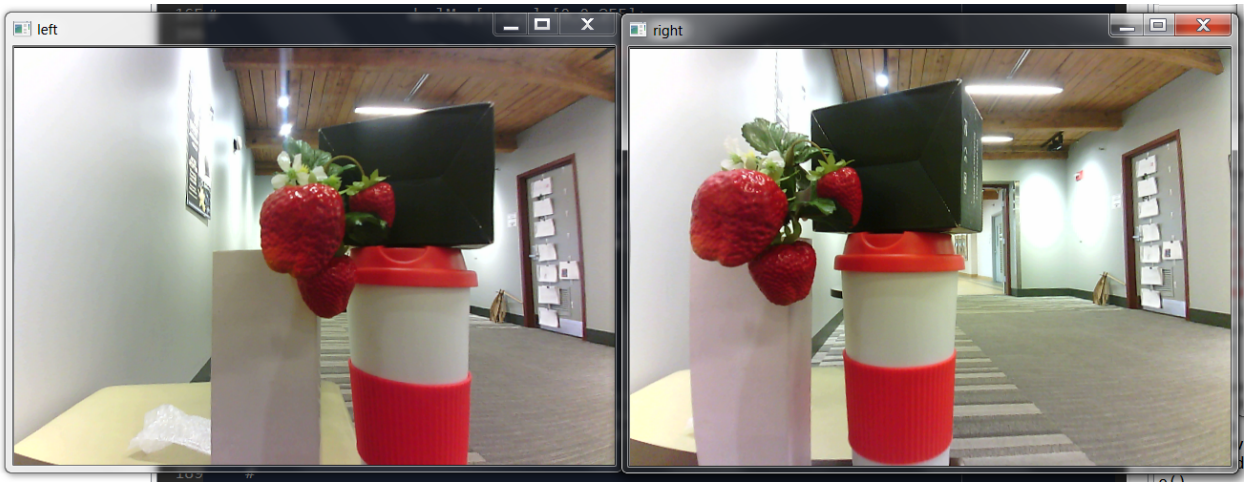

Figure 7. An example of the stereo webcam input that our system receives. The offset in distance between our cameras' mounting points gives us slightly different images from right to left.

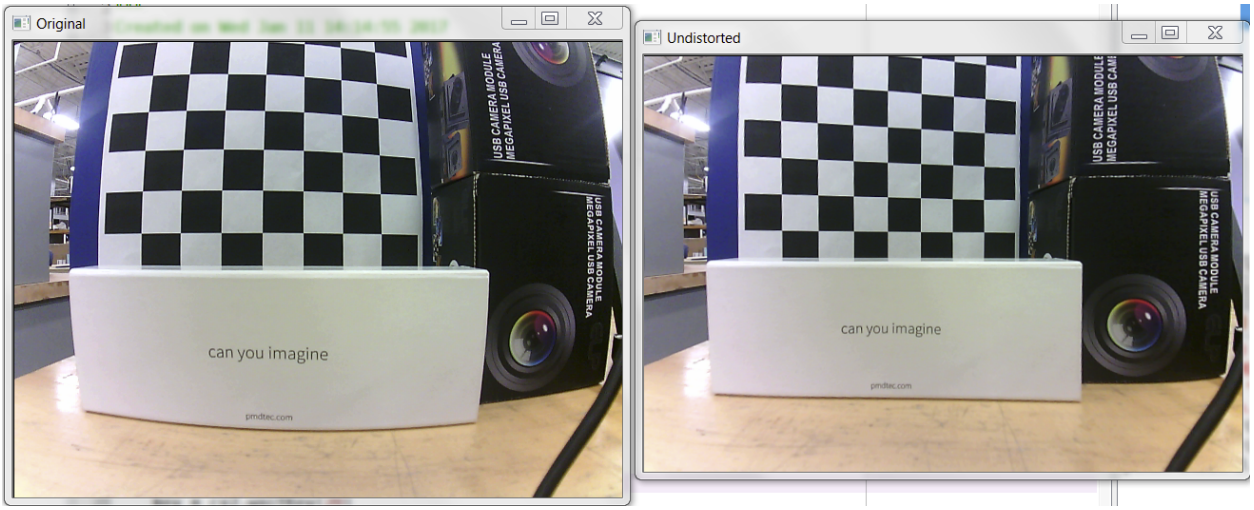

Figure 8. Comparison of the raw webcam image (left) and undistorted webcam image (right).

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 the loss of pixels, bringing the resolution of the video feeds from 640x480 to 629x438, an amount that we determined to be acceptable. With distortion removed, we can correctly use a cartesian coordinate system in our correlation algorithm.

Outputs webcam data.

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

Correlation Algorithm (Correlator.py)

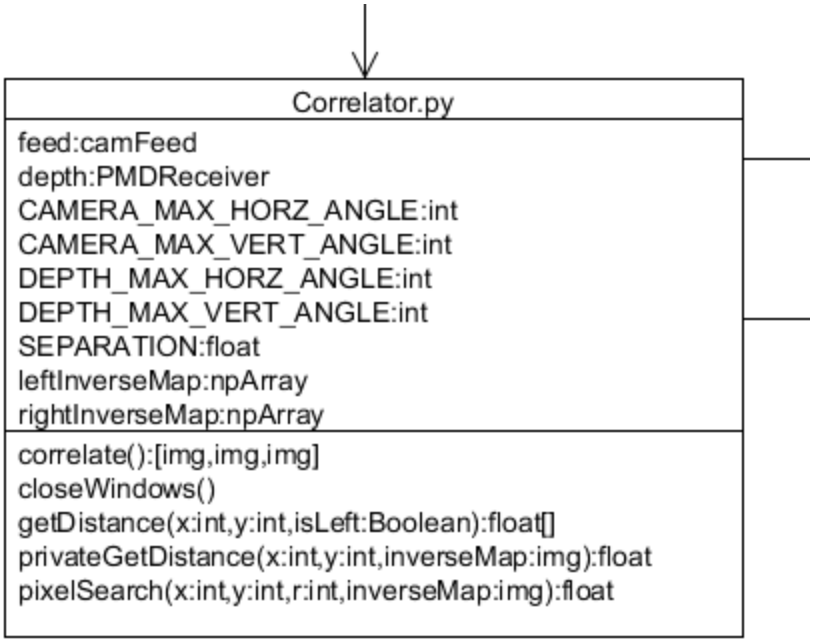

Figure 9. *Correlator.py as shown in our UML*

Import depth camera information and RGB video feeds.

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

One of the issues we observed with the webcams was that one looked slightly more upwards than the the other. This meant that the cameras were vertically out of alignment. We solved this by cropping off the top of the upwards-pointing webcam and the bottom of the downwards-pointing webcam.

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.

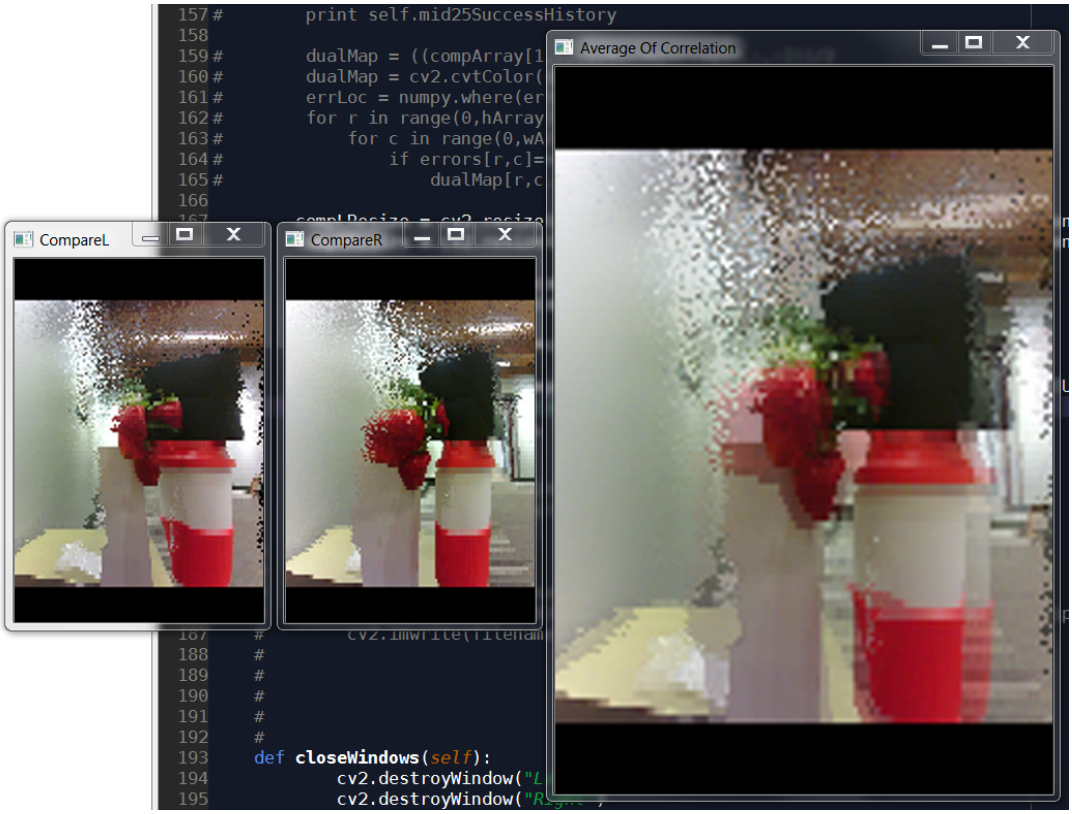

Figure 10. *The depth correlated images in our project. Left: Each pixel of the depth camera has had a pixel from the left-hand webcam mapped to it. Center: Each pixel of the depth camera has had a pixel from the right-hand webcam mapped to it. Right: The average of the other two images produces our 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 10.

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 11. *Representative example of a section of a color image with correlated depth replacing color pixel data. The sparse distribution of distance information is a direct result of the resolution difference between the webcams and depth camera.*

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

Given a pixel coordinate in the webcam and which webcam was clicked on, the getDistance method finds the nearest pixel in that webcam's image that has been correlated to a depth camera pixel. To find this pixel, the program first checks if a depth value was assigned to the requested pixel. If the pixel does not have a depth assigned to it, the method checks if any of the eight-connected neighbors have a depth value. This can be thought of as a square with a "radius" of one pixel that is centered on the given pixel. If none of these pixels has a depth value, a square with a radius one pixel larger is searched. This process is repeated until one or more pixels with an assigned depth value are found in the square, or until the max radius is reached. If more than one depth value is found on the same iteration, the distance value for the pixel is determined by the simple mean of the values. If the max radius of 20 pixels is reached, a distance of 0 is returned. Based on the distance value and direction vector, the method will return the displacement in terms of x, y, z, and the total displacement. Pixels that were tagged as "false" in the C++ code are ignored.

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 below.

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. This code was commented out in our final submission, but could be reactivated easily.

Top Level Script and GUI (GUI.py)

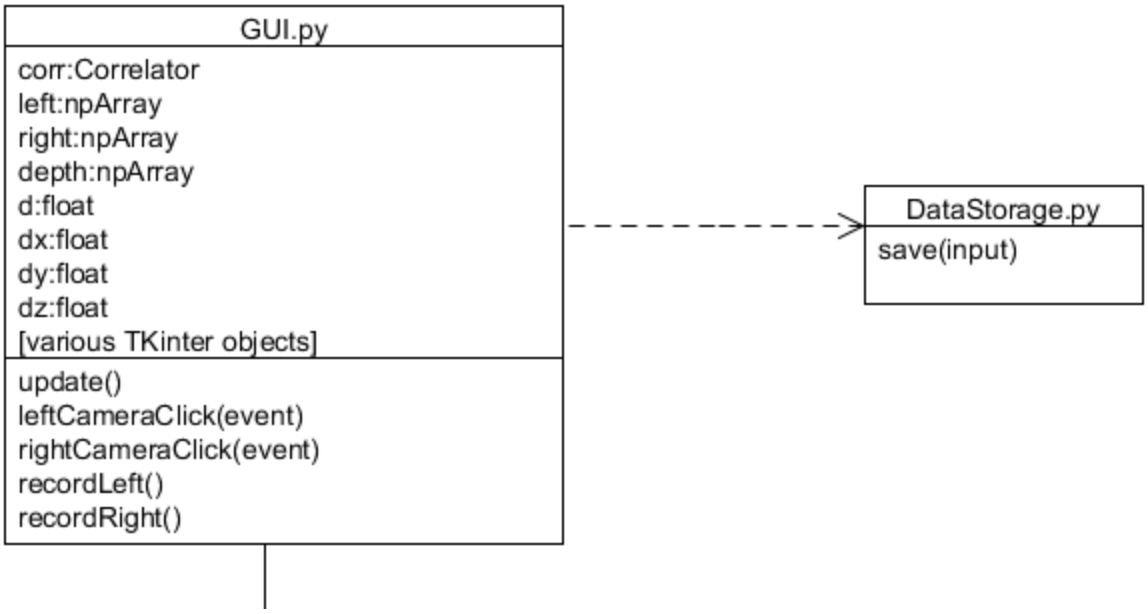

Figure 12. *GUI.py and DataStorage.py as shown in our UML*

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

Shows raw data input from our sensors to the user's screen in a single window. False pixels from the depth camera are shown as black.

Set up click listener for both left and right webcam feed

This script creates a click listener for the GUI showing the left and right webcam feeds. When the user clicks in these feeds, the listener will pass the coordinates of the click and which window was clicked into Correlator's getDistance method. The getDistance method described above then returns the delta x, delta y, delta z, and straight line distance to the clicked pixel as determined by the correlation.

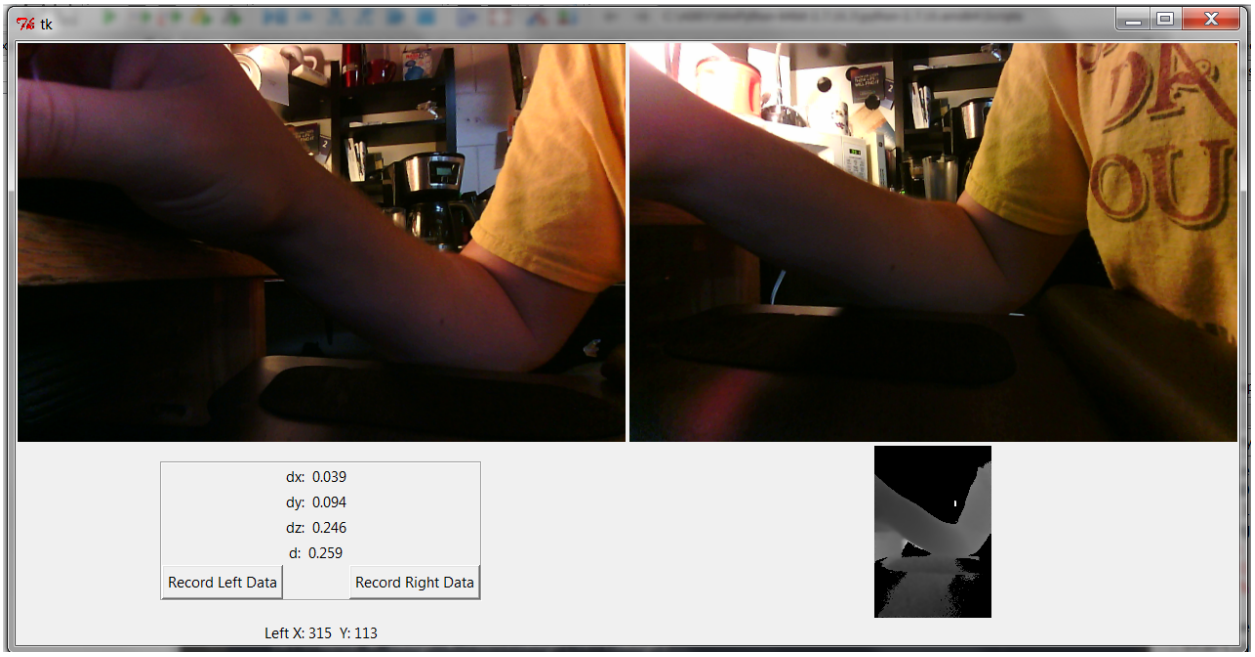

Figure 13. *GUI. The top two pictures are calibrated webcam feeds. The bottom right hand picture shows the depth camera feed with the false pixels set to black. Bottom left is a panel that displays information about the pixel a user has clicked on and holds the data left and right data storage buttons.*

User may choose to save this moment's data.

The user may click a left or right button below the feeds to save this moment. Clicking the left or right save buttons will send the respective RGB camera's current image into the save() method of DataStorage.py. This method saves the webcam image as a .png. At this point, DataStorage.py uses OpenCV and NumPy in the HSV color space to create a binary mask of strawberry colors. We perform a morphological open (erode, then dilate) to clean up the image, then perform a connected components command to find the centroids of each remaining object. The strawberry centroids and the current time are output to a .txt file.

	Hue	Saturation	Value
Upper Bound	5	50	0
Lower Bound	160	255	255

Figure 14*.The hue, saturation, and value thresholds used to identify strawberries in view. Hue values wrap around through zero to 179. No value constraints were set because the robot may be used during nighttime. These values may need to be adjusted for optimal performance in specific lighting conditions.*

Performance Status

Correlation

- The correlation algorithm, while not perfect, works well. As seen in Figure 15, there are some issues with left-right alignment of the overlay of the color pixels onto the depth image. We did not have time to fully diagnose the source of this discrepancy, but we suspect that it was caused by the adjustments made to the frame size, which affects the max view angles of the cameras.

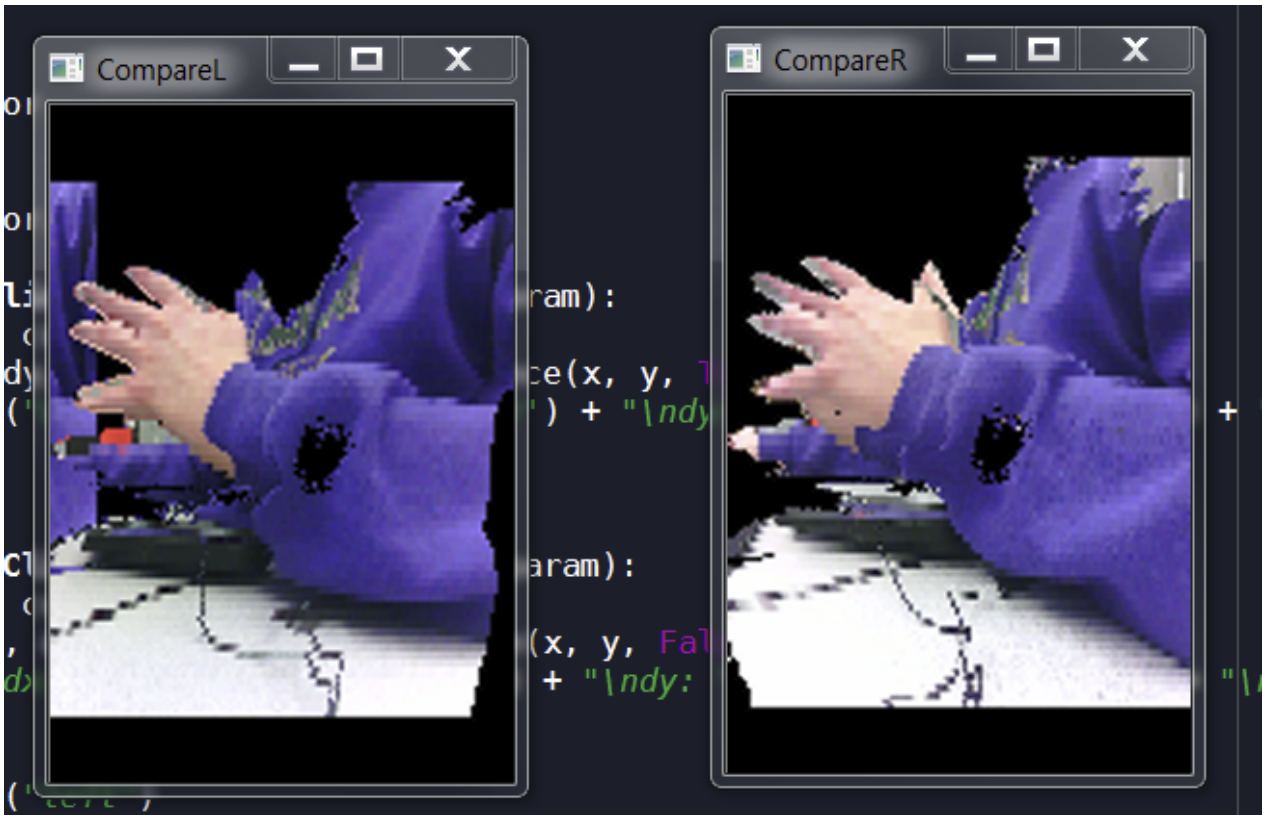

Figure 15: *Approximately half a knuckle of the splayed hand above is cut off because the depth information is shifted to the right of the RGB image.*

Data Storage

- The data storage system recognizes objects in the hue and saturation range of a strawberry in more than one lighting condition. The system was tested in conditions of daylight (100,000 lux) and overcast skies (2,000 lux). In both scenarios it was able to find the strawberries in its field of view, but this performance may vary depending on the exact conditions and camera used.
- All objects that remain after image processing are correctly recorded in a .txt document.
- The system consistently adds a phantom strawberry at pixel coordinates 311, 235.5.
- It is worth noting that this system does not account for several problems that it could encounter in the field.

- The system cannot account for objects obstructing the view of the depth camera, but not a webcam.
- The system is unable to distinguish between two overlapping fruits.
- The system will seeing two distinct fruits where a single strawberry has a vine crossing in front of it.
- The system will not accounting for the size it should expect a strawberry to have at a given distance. For example, a red pickup truck in the background will be counted as a strawberry, even though it is out of the depth camera's range.

Mount

Presentation of Design:

A major addition that was made to our project's scope was to design and build a mount to hold the cameras we used for our project. This prototype holds a webcam on either side of the depth camera. This makes the mount compact and since all three lenses are in horizontal alignment, it is easy to write code for. The drawings can be found in the appendix, but a picture of the SolidWorks model can be seen in Figure 18.

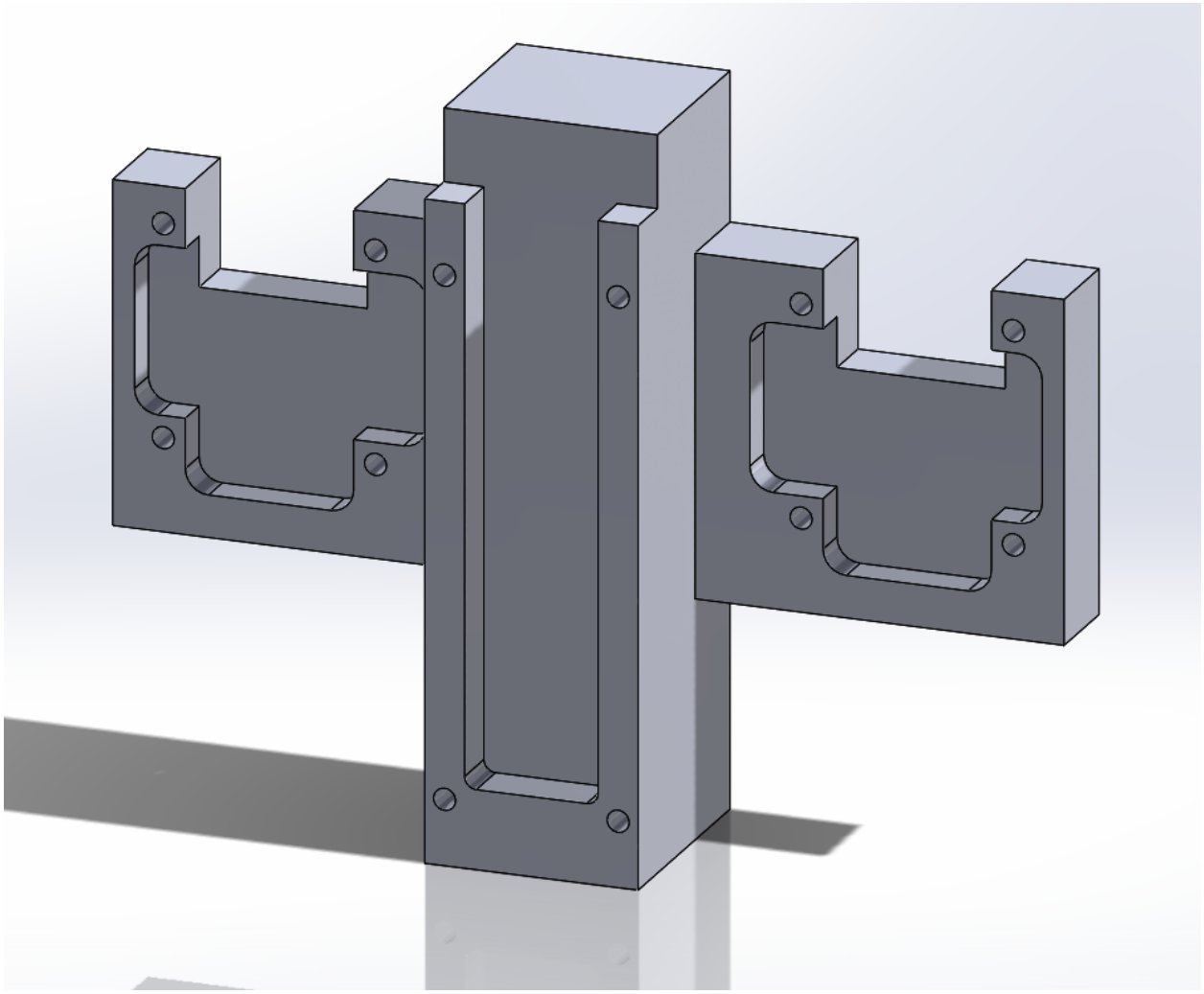

Figure 16. *SolidWorks model of the aluminum mount*

Each wing of the mount has four holes to attach a webcam. A basin has been carved behind it to allow for the camera's circuitry and an open area at the top gives room for the camera's cables. A rectangular face plate (see Appendix) screws into the center of the mount to hold in the depth camera. Extraneous areas of the mount have been removed to lower the part's overall weight.

Performance Status

We have designed and machined an aluminum mount to hold ADEV's cameras and meet our needs. Most of the milling, including the critical dimensions was performed with a CNC mill, but later operations were done by hand. This included the removal of excess material, the drilling of holes, and tapping for screws. The mount holds all three cameras in a horizontal line and is light enough and effective enough at dissipating heat to fulfill our needs. The team considered adding thermal paste to the depth camera's slot to increase the flow of heat from the camera into the mount. We found that this was unnecessary, but if heat flow becomes a problem in the future, we recommend that ADEV considers applying thermal paste to the back of the depth sensor. Below, we iterate over each of the mount's needs and how we met them.

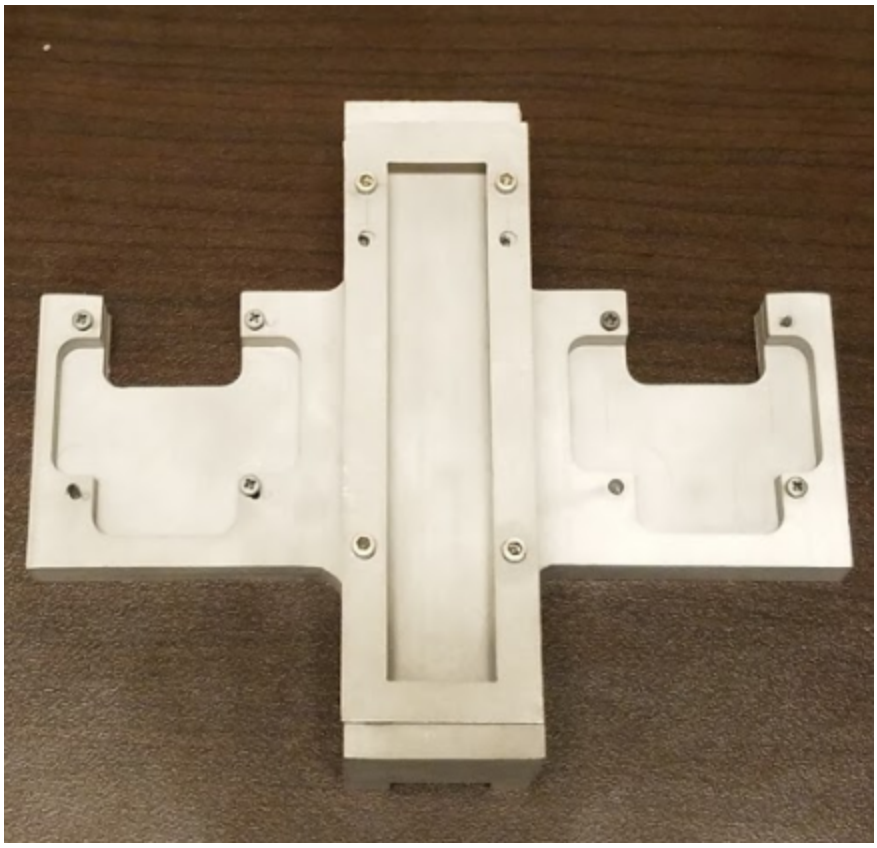

Figure 17. *The aluminum mount made for this project. Webcam slots are on either side of a vertical slit for the depth camera. A faceplate to secure the depth camera is screwed into the front.*

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

The heat sink was made from aluminum 2024, which has a thermal conductivity of 205.0 W/mK. A heat transfer model found that the depth camera would reach a steady state temperature of

80.33°F. In a 90°F - a temperature we would expect from summertime on a strawberry farm. We measured the steady state temperature of the camera to be 3 to 4 degrees higher than the surrounding air.

2. The mount is made from an easily acquired material.

Aluminum bar stock is found on McMaster-Carr's website in a variety of forms.

3. The mount will securely hold our time-of-flight sensor.

The sensor remains in the mount during operation of the robot. This can be tested by placing the sensor into the mount, then rotating it through angles past ones that the robot's arm will go through while picking strawberries. When the face plate's screws are tightened, the camera will not fall out its slot even if the mount is fully inverted and shaken.

4. The mount will hold the sensors in alignment.

The CNC mill used to carve out the mount had a tolerance of ±2 thousandths of an inch. The manual mill we used for drilling the webcam holes had a tolerance of ±5 thousandths of an inch. We considered these tolerances to be negligible.

5. The mount must be light enough to be supported by the robot arm.

The mount weighs 0.69 lbs. ADEV has said that this is light enough for their servos to support.

6. ADEV has enough area to make an attachment.

A 1 inch by 1.25 inch rectangle has been left at the top and center of the mount. This gives ADEV sufficient space to attach this mount to the robot arm.

7. Mount is aesthetically pleasing.

The mount was sandblasted to a smooth, matte finish.

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

The Appendix is available through GitHub. Follow the repository link at the beginning of this report. This report and our appendix may be found under the “Spring End of Quarter Report” folder.

Under the “Appendix” directory, there are several sub-folders. “Project Code” contains the main project’s code. GUI.py is the top level script, but the program will likely fail if the cameras are not attached to the computer. “SolidWorks” contains the SolidWorks files for the mount and faceplate. “Miscellaneous” contains other project documents such as our UML diagram created using Umlet and our heat transfer analysis..