CV Window Detection and Penetration

Vishnu Sashank Dorbala
University of Maryland
vdorbala@umd.edu

Tim Kurtiak
University of Maryland
tkurtiak@umd.edu

Ilya Semenov
University of Maryland
isemenov@umd.edu

Surabhi Verma
University of Maryland
sverma96@umd.edu

***Abstract*—This report presents the implementation and results of a window detection algorithm used for quadcoptor navigation through said window. Color thresholding, edge detection and pose estimation methods were executed to identify and navigate through a yellow and a purple window.**

I. Problem Statement

This project aims to implement an algorithm which used an on-board camera to detect and navigate through a purple and yellow window. The window dimensions are specified below in Figure 1 .

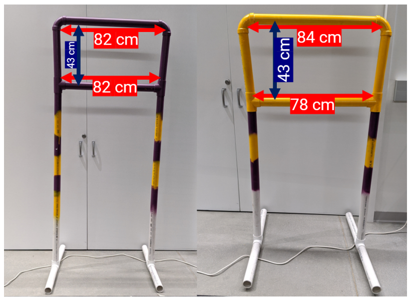

Figure 1. Window Dimensions

The quadcopter must identify the window, estimate its distance and pose relative to it, and then fly through the window. This task requires significant coordination between computer vision, position estimation, and the aircraft controller.

II. Color Thresholding

Color thresholding is used to isolate the yellow and purple windows within the on board camera image stream. Color thresholding works by identifying pixels within the image that match a sample set of training data which has been hand selected. Several methods with varying levels of complexity can be implemented to achieve color thresholding.

Manual Thresholding: Manual thresholding simply works by specifying a range of acceptable pixel values and checking the value of each image pixel against that range. Equation 1 below shows an example of manual thresholding in an RGB color scheme.

$$\begin{cases} R_{lower} < R < R_{upper} \\ G_{lower} < G < G_{upper} \\ B_{lower} < B < G_{upper} \end{cases} \tag{1}$$

The manual thresholding approach specifies a bounding box in the color space where values within the box are accepted. However, the bounding box often does not fit the sample data well and may include colors which are not intended as the target color. In order to remedy this, a custom color space can be generated by rotating the 3-dimensional RGB color space to allow a bounding box to better fit the data. However, the limitation of manual thresholding to a course box limits the usefulness of this method and leaves much to be desired.

Single Gaussian: Statistics provides a useful tool, the multivariate normal distribution, which can be used in color thresholding to represent a set of sample data as a probability function. Below in 2, we use a Gaussian distribution in 3 dimensions to represent the probability that a pixel is in the training data set.

$$p = \mathcal{N}(\mathbf{x}, \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^3 \ |\Sigma|}} e^{-\frac{1}{2} e_0^T \Sigma^{-1} e_0} \tag{2}$$

Where p represents the relative probability that the pixel is within the sample set, Σ is the empirical covariance of the training data and e_0 is the error between the sample pixel $\mathbf{x}$ and the training set empirical mean μ. While in this form p does not have a physical significance of probability, it can be used as a relative measure of fitness of the data to the target distribution.

In practice, the probability value for each pixel in an image is calculated and normalized to the maximum probability according to the equation below:

$$p_{norm}(x, y) = \frac{p(x, y)}{\max(p)} \tag{3}$$

where x and y represent x and y pixel coordinate position within an image.

A threshold T is defined as a value between 0 and 1 and used in conjunction with the model above to filter values which best match the single Gaussian model.

The single Gausian model is computationally fast and does a good job of representing simple sets of training color data. However, it will have difficulty representing more thorough datasets where lighting conditions change.

Gaussian Mixture Model: In order to better fit complex training color datasets which may not be well represented by a single Gaussian model, a weighted sum of multiple Gaussian models may be used. Simply:

$$p = \sum_{i=1}^{K} \pi_i \mathcal{N}(\mathbf{x}, \mu_i, \Sigma_i) \tag{4}$$

Where K is the number of Gaussian model clusters used to represent the data. When generating a Gaussian mixture model, it should be noted that a higher K will result in a more computationally expensive solution and may slow down real time applications.

Generating a GMM can be executed through an iterative process called Expectation Maximization. First, the Gaussian Mixture Model is initialized with a set of random means and a random positive semidefinite covariance matrix. It is best to initialize the covariance matrix to a large value to begin with, otherwise the model can encounter a divide by zero error in the first step.

The E-step evaluates the model correctness at its current state. To do this, a cluster weight is assigned according to Equation 5 below.

$$a_{i,j} = \frac{\pi_j p(\mathbf{x}_i|C_j)}{\sum_{j=1}^{K} \pi_j p(\mathbf{x}_i|C_j)} \tag{5}$$

where i is the data point in the training set and j is the GMM cluster index out of K clusters. The values for a represent a relative goodness of fit for the cluster model j to data point i.

The M or Maximization step uses the results from the E step to reset the GMM cluster values to better fit the data. The following updates to the mean, covariance, and weighting factors are executed.

$$\mu_j = \frac{\sum_{i=1}^{N} a_{i,j}\mathbf{x}_i}{\sum_{i=1}^{N} a_{i,j}} \tag{6}$$

$$\Sigma_j = \frac{\sum_{i=1}^{N} a_{i,j}(\mathbf{x}_i \mu_j)(\mathbf{x}_i \mu_j)^T}{\sum_{i=1}^{N} a_{i,j}} \tag{7}$$

$$\pi_j = \frac{1}{N}\sum_{i=1}^{N} a_{i,j} \tag{8}$$

The process is then iterated until the models converge. Simple convergence can be determined by observing the change in model means from the previous step to the current step. However, the convergence of the overall GMM mean does not have much significance since one part of the model moving in the opposite direction of another may result in a net neutral model mean. For this implementation, it is recommended to judge convergence by setting a small threshold on the change in model weight factor $\sum_{j=1}^{K} \pi_j$. This convergence criteria guarantees that the model has settled to a steady state. Model convergence may be computationally expensive and take some time depending on the initial guess.

GMM solutions are typically very good, but some clusters may be less beneficial than others. For example, a GMM with a weak cluster weight which is an insignificant percentage of the total cluster weights may not be benefiting from the number of total clusters. The weak cluster could just as easily be removed and result in no loss of model fidelity. It is recommended to review output weighting factors for extremely small weights and consider reducing the GMM cluster dimension if such clusters are found.

The implementation used in this report uses a cluster value of $K = 3$ in order to benefit from the increased robustness of GMM while maintaining a minimal computation time. However, computation time was still too slow with GMM even after reducing image size to 20%. As a result, a compromise was made to use a well tuned single gaussian for thresholding instead of GMM in order to prioritize run time.

Sample yellow window thresholding results for GMM and single gaussian are included below in figure 2 . The results clearly show that both GMM and single gaussian are able to identify the window in low light. This particular image also suggests that the single gaussian performs better than GMM, however the true measure of performance can only be demonstrated in multiple angles and lighting conditions.

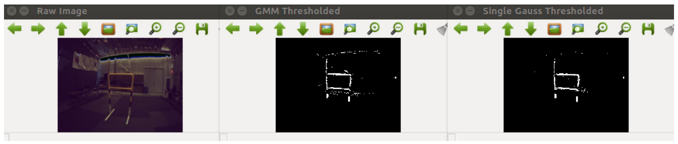

Figure 2. GMM and Single Gaussian Thresholding

Finally, the training sample set in RGB space is shown below in 3

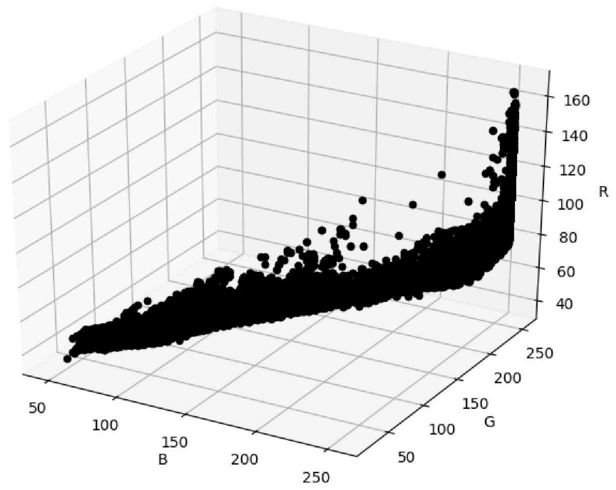

Figure 3. RGB values in color space

III. Vertex/Edge Detection

Detecting vertices was accomplished through a procedural iterative process, with many explored options. The final result is both robust to the most inconvenient of noise, and relatively fast to execute. There is room for improvement in continued optimization, as the process time is not insignificant, and better rejection of false positives.

The vertex detection is a 6 step process: basic thresholding, gaussian thresholding, mask closing, edge detection and filtering, line application, and finally vertex location. Each step will be explained below.

Figure 4 below shows a low light raw camera image which is difficult to process.

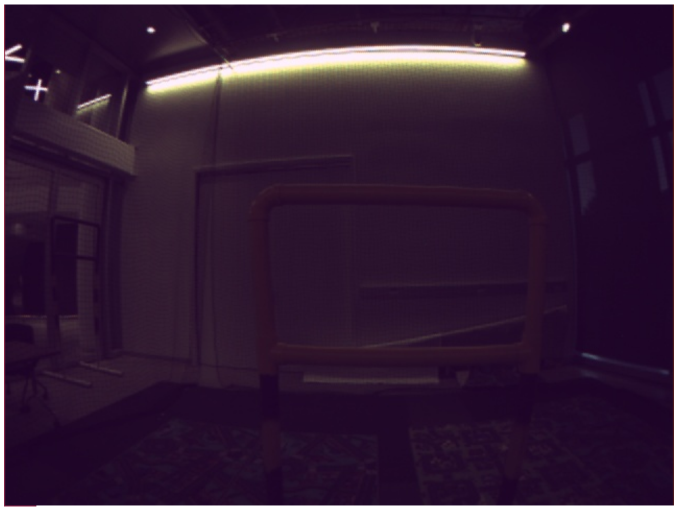

Figure 4. Sample Image

The first step is to use basic RBG thresholding. First the image is passed with a 3x3 kernel median blur, then a 5x5 kernel median blur. This smooths out colors within edges without distorting the edges too much. Next, two points were selected in pre-proccesing that were a part of the gate in the highlight and shadow in order to apply a course manual threshold. These are chosen as the upper and lower thresholding values respectively, and a padding is added to both. This purposefully creates an over selective mask, see figure 5. This significantly reduces the number of pixels which additional operations must be performed.

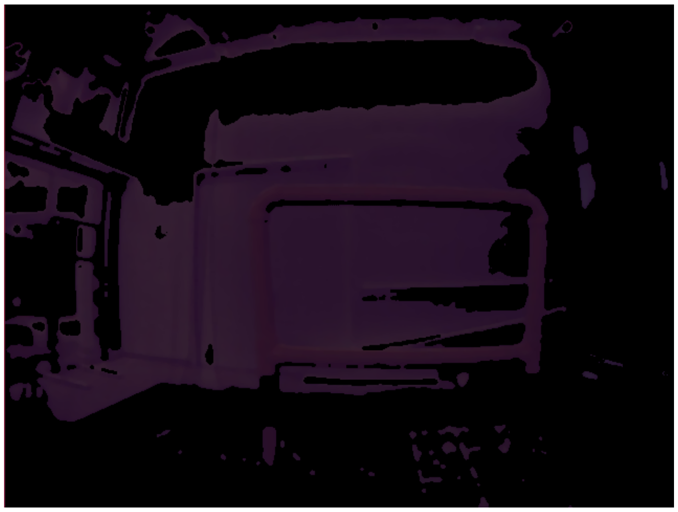

Figure 5. Masked Image

The next step is to use this image to apply a single Gaussian filter. The result is a mask with minimal error as shown in figure 6

With a suitable mask achieved, a series of dilations and erosions is performed to close any parts of the mas that are close together, and remove small noise away from areas with a high concentration of pixels that match the yellow window. The result from this is a blocky series of shapes depicted in figure 7.

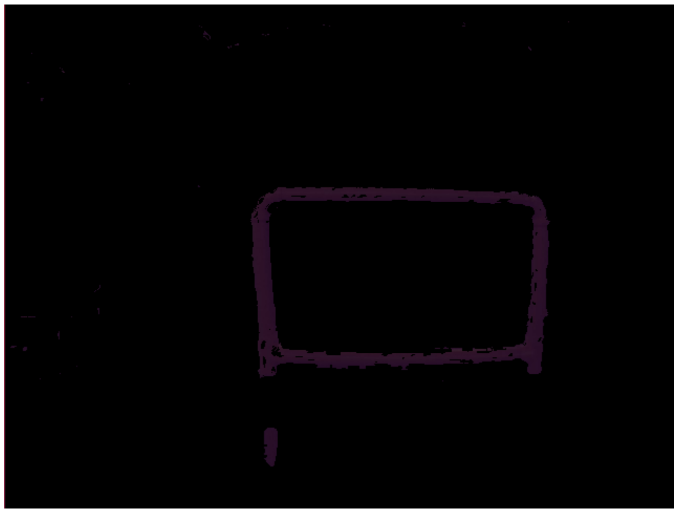

Figure 6. Extracted Mask

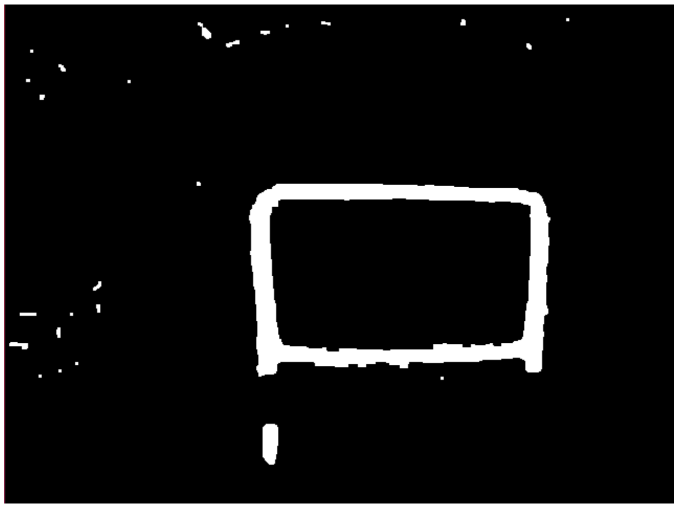

Figure 7. Mask after Dilation and Erosion

This allows us to preform one of the most critical steps in the vertex finding process, contour fitting. A contour function fits a line that describes the perimeter of the shapes found in this dilated mask. These contours represent important edges, and the ability to draw them allows us not to have to use a Canny edge detection function. Contour finding is relatively expensive, but allows for filtering of noise that tends to cause difficult to filter errors in other implementations.

The largest contour by enclosed area is assumed to be the window. Rarely will there be a continuous set of noise in the mask after dilation and erosion operations that is larger than the window. This allows us to simply and immediately disregard any countours whose centroid lies far outside of the largest contour.

Furthermore, because the contours themselves are edges they replace the need to use and tune Canny function parameters. Instead, it is simple to select the contours whose length is some acceptable fraction of the length of the largest contour. This selection based on area allows us to account for distance in the selection of acceptable edge lengths. Consider an image that displays a window far away, its area will be small relative to the area of a contour of a window that is close by. The perimeter of the windows edge may not be continuous due to noise, and because of distance, the pixel length of potentially crucial edges of an image depicting a distant window may be comparable to the length of edges depicting random noise in other images depicting a window near by. It can be very hard to pick the acceptable edge length using other methods before having some metric related to the distance of the window from the camera. Contours allows us to expect some range of edge lengths in pixels before having a good estimate of window pose relative to the camera.

This technique also accounts for noise that occurs very close or inside of the pixel space depicting the window. One of our earlier attempts at vertex detection used contours to mask Canny edges near the largest contour. This approach was effective in reducing noise well outside of the window's pixel position, however, noise that occurred within or near the window's pixel position was kept in this mask. Any such noise in the color thresholding mask that occurs behind the window, and therefore within the window's pixel space, had major effects on the estimated location of the vertices of the window in this more traditional Canny based algorithms.

Once the filtered contour edges have been selected they are drawn and look like what is shown in figure 8.

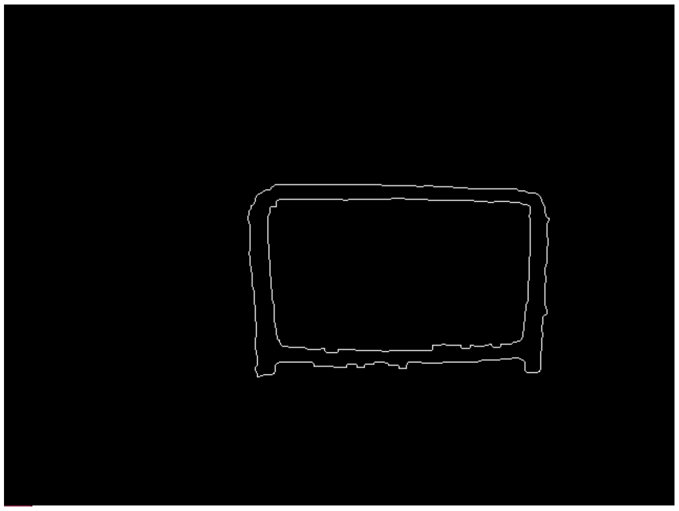

Figure 8. Relevant Edges of the Mask

Lastly, a probabilistic Hough line transform is performed on these edges. This method is less computationally intensive than a standard Hough line transform and allows for greater tuning of acceptable lines based on contour perimeter properties found earlier. Additionally, the calculation of infinite line intersections in point slope form only involves algebraic division, multiplication, addition and subtraction, whereas the intersections in polar form involve the calculations of sines and cosines. The lines given by the Hough transform are split up into near horizontal and near vertical lines, all others are discarded. Then only intersections between these near horizontal and vertical lines are found rather than all intersections since these are the most relevant. Refer to figure 10 for a depiction of the lines a probabilistic Hough transform returns. Note that these lines are interpreted as infinitely long when performing intersection operations.

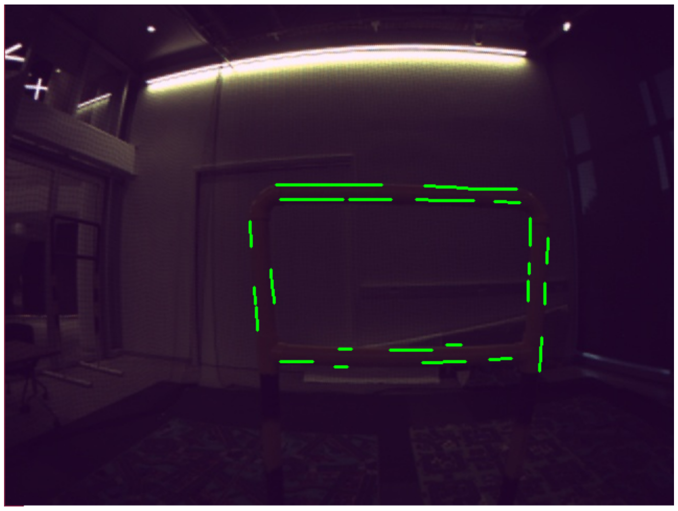

Figure 9. Relevant Lines of the Mask

The maximum and minimum x and y pixel coordinates of the intersections are used for several purposes. First, they are used to find the overall aspect ratio of the image feature, if this ratio is far outside of acceptable values for the window, then it is concluded that these features are noise. Additionally, if intersections are found near the midpoint of the extremea, they are discarded as no relevant corners would exist close to the window's center. Finally, the midpoint of the extremea is used as the intersections of 4 quadrants, allowing the sorting of all intersections into groups related to each of the four corners of the window.

Once the intersections are sorted, the ones that are closest to the midpoint in each quadrant are labeled as the inner corner points, and the furthest ones are the outer corners. The results are shown in 10 below.

This completes the procedure for locating vertecies of the window. It has advantages in noise filtering and is fast compared to other methods with equal performance. The downsides are the tendency to get outer corners that are further out than the window corners due to the slightly trapezoidal shape, and curved edges. However, inner corners are more accurate, and only they are used for pose estimation.

IV. Pose Estimation

In order to estimate pose from vertex points of a known window, we use the Opencv PnP algorithm which gives us the pose $([r_w^c | t_w^c])$ of an arbitrarily defined world frame (w)

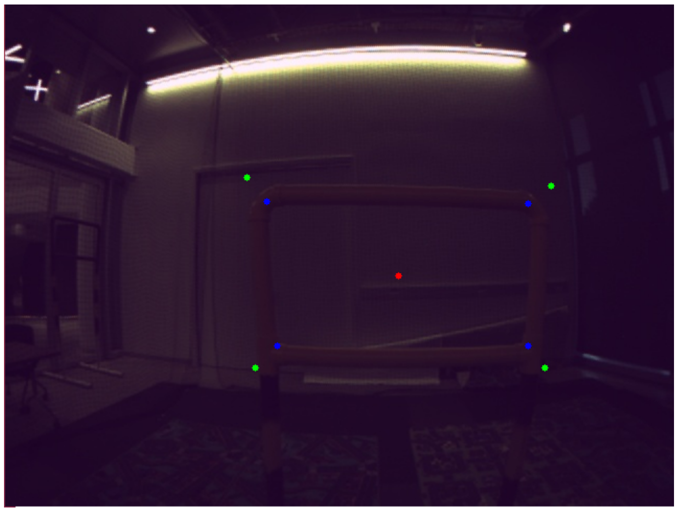

Figure 10. Output Corners of the Window

relative to the camera (c) frame attached on the bebop. We define the world frame to be attached to the center of the window. This is the point where we would ideally like the drone to reach and go past it. However, ROS accepts commands in terms of velocities, along and about the body frame, to move the robot to a particular point, rather than the location of the point itself. So there are three frames into play here, the world, camera and body frame.

We have the following from Opencv's SolvePnP:

- The rotation vector of w wrt c: r_w^c
- The translation vector of w wrt c: t_w^c
- The rotation matrix of w wrt c: $R_w^c = Rodrigues(r_w^c)$

From this we calculate:

- The rotation matrix of the body frame wrt camera frame:

$$R_b^c = \begin{Bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{Bmatrix}$$

- The rotation matrix of the world frame wrt body frame: $R_w^b = (R_b^c)^T * R_w^c$
- The translation vector of the world frame wrt body frame: $t_w^b = (R_b^c)^T * t_w^c$

We determine the body frame relative to the world frame:

- The rotation matrix of the body frame relative to world frame $R_b^w = (R_w^b)^T$
- The translation vector of the body frame relative to world frame $t_b^w = -(R_b^w)^T * t_w^b$

This allows us to place way-points in the world frame relative to the window, and translate them into the body frame as x,y,z and yaw desired poses. These poses are passed to the controller function as the desired vector in the body frame.

V. Implementation

Our approach towards implementing the window passing task consists of two stages. First, we use techniques in Classical Computer Vision to perform operations for obtaining "key points" from a camera sensor image. We obtain these points using colour thresholding, as well as using a Gaussian Mixture Model. We also implement a single Gaussian filter that thresholds image colours based on the These points (4 inner corners, 4 outer corners, and the center of the window) are then used to solve camera extrinsic that give us the orientation and position of the target window in the 3D world frame, with respect to the 2D image frame. This information is used to identify desired change in position and orientation in the body frame which is passed to the controller.

The controller is a simple heuristic P controller on the desired pose. The gains on the vector and yaw orientation are tuned and a few heuristic statements are accounted for:

- If no corners/valid pose estimation is found, yaw in the direction of the largest color thresholded mask contour, which implies yawing toward the window if only a portion of it is seen.
- If the desired vector has a large z component, perform only z adjustments first, this alleviates the wide difference in gains found between z and x,y motion commands as well as increases likelihood that the window corners will be seen throughout the maneuver. This effectively prioritizes getting in plane with the window before translating toward it.
- If no new desired vector has been acquired, perform a z translation in the direction of the last desired vector's z component. This accounts for issues where the drone is too close and too far above or below the window to get an accurate estimate for position

VI. Results

Thresholding result videos are posted here: https://www.youtube.com/watch?v=E15kOavVWCg

Tim notes:: we flew thre one okay, but had issues with all the others. one of the main problems is that the pnp estimates are worse the farther away you are, but also result in more erratic approach paths. By the time you get close enough to get good cosnistent readings youre fucked with how much you can or can not see and shit. Tuning to lighting conditions was rough, as the sun set somewhat by the time we got going, so the filters based on 0% lights or 50% lights or whetever didn't apply anymore. We crashed through a few, thats just not enough stop and gain confidence in your estimate before flying through. idk what else to say

VII. Lessons Learned

The most valuable lesson learned was the importance of a sound controller strategy relative to accurate position estimates. A mistake was made in focusing too much on optimizing tuning parameters for computer vision and accurate PnP solver returns, that the time left for using this data in a a controller was minimal. Aspirations of a proper PID controller with odometry assistance and drift correction via computer vision exist, but time constraints forced a simpler approach. As a result, despite good vertex detection and pose estimation, this information failed to provide adequate value due to the simple controller scheme.

Additionally, the importance of optimized calculations in code were very apparent in this project. The best example is the initial working implementation of a Gaussian filter using a for loop being unusable due to calculation time. Correct output is no longer enough, and solutions optimizing multidimensional array operations needed to be found.

Lastly, the effect of lighting on computer vision performance was very apparent here. Training the filters for a variety of not only light intensities, but also types (outdoor, indoor, directional,etc.) was a major challenge. One solution does not fit all implementations in this case. A more robust masking approach is left to be desired.

VIII. Conclusion

while peak performance was not achieved through this implementation, a few tw