ENAE788M Final Project Team 5 - QDMC

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 development and implementation of computer vision and control algorithms to navigate the PRG Husky drone through a series of obstacles. Challenges in this course include autonomously flying through a window, avoiding a wall, crossing a bridge, and landing on a bullseye. This project was conducted as a part of UMD's ENAE788M Autononomous Aerial Robotics class final project.**

I. Problem Statement

The final project for ENAE788M [1] is to autonomously fly a predefined course consisting of six obstacles as fast as possible. The obstacles encountered are listed below:

1) Avoid and cross over a low wall;
2) Fly through a yellow window;
3) Avoid a river and fly over a bridge;
4) Land on a circular bullseye target;
5) Avoid and cross under a high wall;
6) Land on a square bullseye target;

An image of the course starting at the bottom left of the image and working clockwise is shown in Figure 1. An imprecise

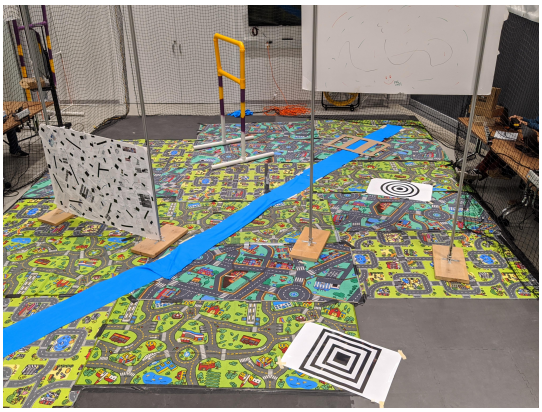

Figure 1: Obstacle Course Overview

prior location of each obstacle and orientation is available prior to flight, but each obstacle is subject to moving prior to the flight attempt. The sections below describe our approach to solving each obstacle. No user input is allowed during the flight, so the aircraft must detect, identify, and navigate each obstacle in succession on its own.

II. Drone Hardware

The hardware for this project was provided by the University of Maryland Perception and Robotics group (PRG) [2]. The quadrotor is largely based on a Parrot Bebop, refitted on to a lighter carbon fiber frame and with several sensor improvements. The list below documents some of the notable hardware improvements:

- Companion computer: Intel Up Board running Ubuntu 16.04 and ROS
- Leopard Imaging Front Facing Camera
- Duo3d Black and White Stereo Camera

An image of the drone is shown below in Figure 2 The

Figure 2: PRG Husky Research Drone

drone is remotely controlled via a wifi network and SSH from a laptop. A user on the ground is required to launch a set of ROS nodes remotely on the Up Board, and the drone navigates the course autonomously from there.

III. Clear the Walls

The section addresses the approach and solution to wall obstacles number 1 and 5 in the obstacle course. The wall obstacles are set up in a predefined low or high orientation in the course. An image of the low wall is shown in Figure 3 below.

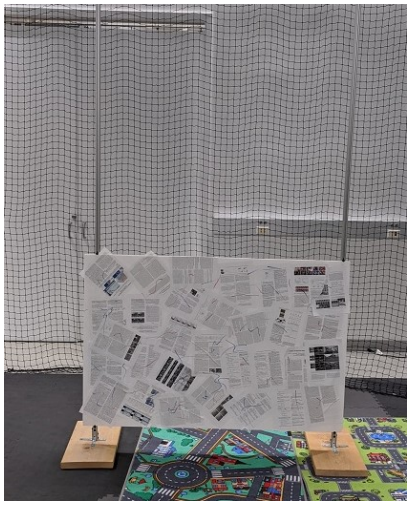

Figure 3: Low Wall Obstacle

A. Detecting the Wall

The team's original approach to wall detection was based on forward camera feature matching of a prior image of the wall to the current frame. Ideally, a set of positive feature matches could be used to calculate the position and orientation of the wall. However, in practice, false positive feature matches from non wall objects caused an erratic solution which resulted in poor performance and an eventual crash into the legs of the wall during the Project 4b demo.

A second, simpler approach was taken to identify the wall for the final obstacle course. Instead of detecting the wall, the forward facing camera instead detects the feet of the wall using Gaussian Mixture Model color thresholding to isolate all wood-colored objects in the frame. This approach is much less prone to errors and works well as long as the feet are in view of the camera. In order to run this method quickly, only the lower half of the front facing camera image is processed, as the wall feet will never be present in the top half of the image.

Additional robustness was added to the feet identification method by removing outliers and isolating only the largest two objects in the image. A set of dilation and erode operations were implemented on the color thresholding mask to reduce false positives and join close together blobs into one object. Next, contours are calculated for the mask using the $cv2.findContours$ [3] function and the results are sorted to yield only the largest two contours, if two exist. An example image is processed below in 4 showing the successive improvement in solution accuracy after each step. In the above

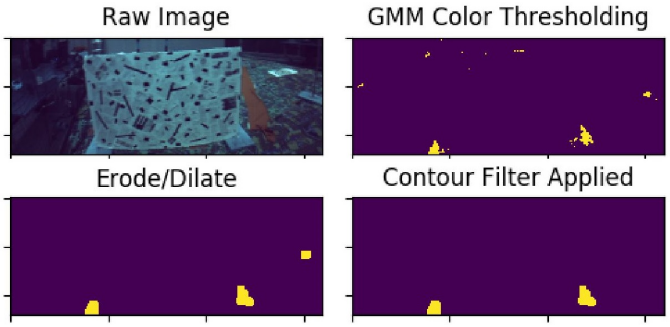

Figure 4: Wall Feet Detection

example, the foot from the second wall shows up in the raw image. However, due to the contour processing step, the second wall foot is filtered out and only the wall in direct view is identified.

B. Wall Controller

After the wall has been identified, the vehicle will need to maneuver through the plane of the wall without hitting the wall or the legs. Due to the nature of the course, the height of the wall is known and thus fly over or under altitude is predefined for each of the wall obstacles. The aircraft must simply line up with the wall and maneuver between the feet at the specified altitude.

To do this, the image coordinate center of each foot contour is determined. If only one foot contour is detected, the quadcopter is instructed to yaw towards the foot in an effort to get a better view of both feet. This works to establish a good view of both feet in the wall before attempting to pass through.

If both feet are in view, the image coordinate center point between both feet is identified. A rolling average across 5 image frames is computed, and if the the center point is stable and the feet are not too close together (¿100 Pixels), the quadcopter is instructed to move 3 meters in the direction of the average center of the feet. This maneuver reliably shoots through the plane of the wall without getting close to the feet.

If the feet appear too close together (¡100 Pixels), the quadcopter takes a 0.5 meter step towards the wall. This action makes sure the vehicle does not attempt to shoot through the wall if it is excessively far away.

The controller as defined is able to handle wall obstacles at close range with ease and speed. However, future improvement could robustify the controller to cases where the wall appears at a harsh angle to the aircraft. However, these edge cases are not necessary for this project as a predefined takeoff point is established for the aircraft prior to each wall obstacle.

IV. Fly through the Window

This challenge involves using a front facing camera to detect and navigate through a yellow window. The window dimensions are specified below in Figure 5 .

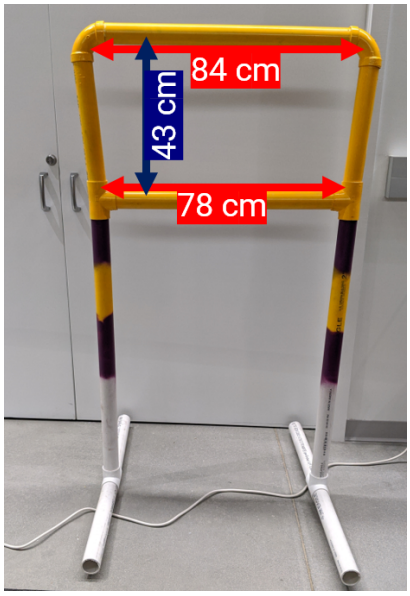

Figure 5: Window Dimensions

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.

A. Yellow Window Detection

The window is detected using color thresholding with a Gaussian Mixture Model GMM [4]. The GMM allows detecting of yellow objects in the camera image in a variety of lighting conditions. 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 6.

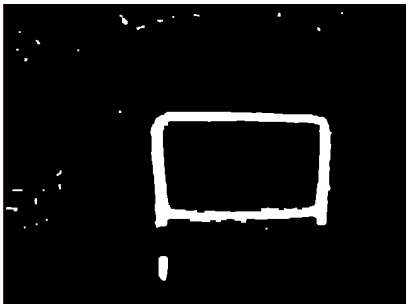

Figure 6: Mask after Dilation and Erosion

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 contours 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 7.

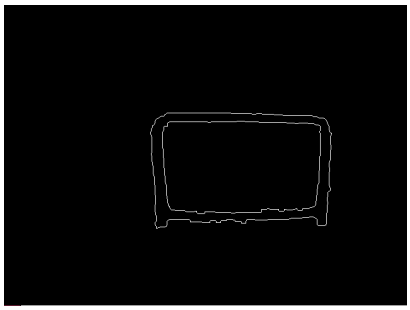

Figure 7: Relevant Edges of the Mask

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 9 for a depiction of the lines a probabilistic Hough transform returns. Note that these lines are interpreted as infinitely long when performing intersection operations.

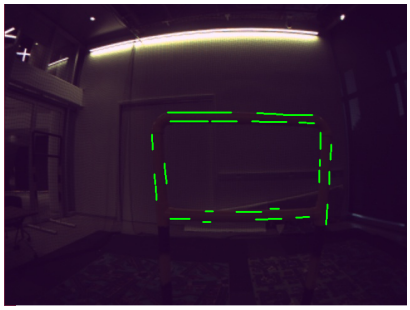

Figure 8: Relevant Lines of the Mask

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 extrema, they are discarded as no relevant corners would exist close to the window's center. Finally, the midpoint of the extrema 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 9 below.

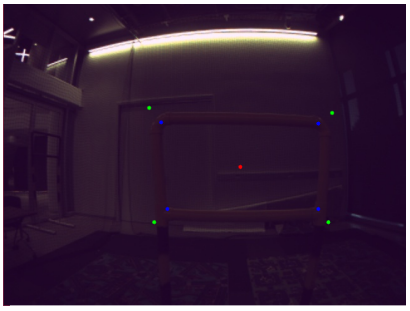

Figure 9: Output Corners of the Window

This completes the procedure for locating vertices 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.

B. Window Controls

PnP is used with the inside corners found in the image frame in the above section, and measured real world positions of these same inside corners. The result is an accurate estimate of the position of the window, and a less accurate estimate of it's orientation. Originally, the controller set waypoints for the drone 1 meter in front of the window, relative to it's orientation. This method is robust against the window's orientation, however, due to the noise in window orientation, and the sensitivity of the 1 meter away waypoint to those angles, the waypoint's position is extremely noisy in z and y (where x is body frame forward and z is body frame up). To alleviate this issue, the controller instead goes to a point that is lined up with the window itself in y and z, and the waypoint in x. The result is that the drone goes left and right such that lines up with the window, but does not attempt to position itself on the window's normal vector. The drone proceeds to navigate to one meter away in x and aligned with the window in z. After this position is assured for some time, the drone flies to a waypoint 1 meter past the window. Then this portion of the race is complete.

V. Fly Over the Bridge

The river and bridge must be flown over, prompting the use of the down facing Duo camera. Since this camera is grayscale the identification of the river and bridge cannot take advantage of color. Additionally, this is a flat feature so stereo identification is also not helpful. Alas, we turn to texture based methods.

A. Law's Texture Energy Masks

The first attempted implementation made use of Law's texture energy masks. These masks are a a series of 2D convolutions that create, after some filtering, 9 variables describing a texture aspect for each pixel in the image.

Shown in figure 10 are the fundamental vectors that can create 16 possible combinations of matrices. These matrices are convolved with the image to create a 16 element vector at each pixel. The vectors corresponding to inverted convolutions are averaged together for rotation invariance (i.e. L5R5 and R5L5).

```
L5  =  [ +1  +4  6  +4  +1 ]  (Level)
E5  =  [ -1  -2  0  +2  +1 ]  (Edge)
S5  =  [ -1   0  2   0  -1 ]  (Spot)
R5  =  [ +1  -4  6  -4  +1 ]  (Ripple)
```

Figure 10: Law's fundamental vectors

These remaining 9-d vectors can be clustered with the k-means method. For robustness, the number of clusters is found by using the elbow method with a random subsample of the entire image for speed.

Looking at figure 11, we can see a grayscale image with various objects that would be difficult to discern with thresholding alone. However, after applying Law's energy masks we can see the image has neatly broken down into 4 base textures, shown in figure 12.

Figure 11: Original image pre-textures

This method was discarded eventually due to the large computation time involved in taking 33 convolutions of the entire image.

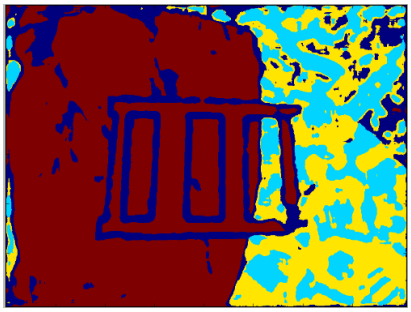

Figure 12: Laws textures result

B. Min-Max Differential Filter

This idea was inspired by Animesh from team sudo. A custom 2D convolution takes the difference between the maximum and minimum values of a window, and uses that as the value of the pixel at the center of the window. This creates a map of featured areas. Due to the intensely varying carpeting, and relatively uniform surface of the bridge and river a mask of only the minimum values from this differential filter returns regions of interest as apparent in figure 13. Unfortunately there are other areas with few features in the netted area that are included in the mask, however, they tend to be darker areas as well. Because the bridge has a light color in the grayscale image, a bitwise AND operation of a simple threshold for higher values along with this differential filter low value mask returns the bridge and river reliably. This AND operation is shown in figure 14

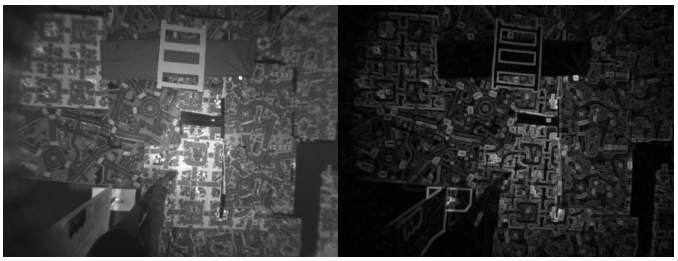

Figure 13: Differential filter of the image on the left is on the right

Figure 14: Regular thresholding on the left, Differential threshold in the middle, and resulting mask on the right

C. Poor Man's Differential Filter

This idea was inspired by Nick from team sudo. Due to the long processing time of a true differential filter, a similar result is obtained by using a liberal Canny filter, and utilizing only the areas with few edges. The resulting mask is noticeably worse but, much faster as you can imagine from the base output shown in figure 15.

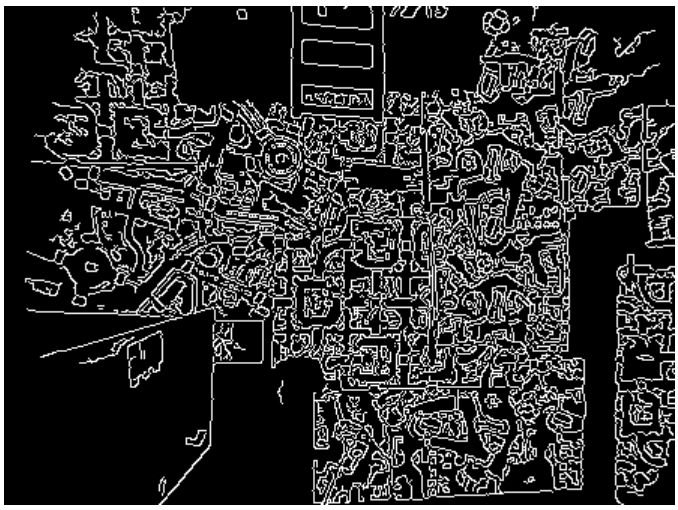

Figure 15: Example of a liberal canny result, morphological operations here can approximate the differential filter mask

This approach is combined with a custom double-threshold filter to produce an adequate mask. One issue is that even with the IR LED that the duo3d camera uses off, there is still a noticeable bright spot in the center of it's vision. The double mask uses two sets of thresholds for the inner and outer areas of the image to account for a drop in bridge brightness out of frame, without creating masks that are too inclusive.

The AND operation as before is used here as well.

This mask can be used with morphological operations to obtain the largest contour. The largest contour, assumed to be the outer perimeter of the bridge, is used to additionally find the holes in the bridge by filtering on: relative perimeter, distance to maximum contour, solidity, aspect ratio, and parallelism. These inner holes tend to lend well to accurate rotated rectangle fits, which allows the calculation of the line that crosses the bridge in the correct direction. The center of the bridge is assumed to be the center of the largest contour, A.

D. Bridge Controller

With the line and a point found, one can assume that the vehicle lies at the center of it's own image C. Therefore the line segment representing the minimum distance to the vehicle from the line AD can be found and used to inform the controller.

If this line segment distance is long, then the vehicle is at a skew to the bridge and should navigate to this line by attempting to find point D. If this distance is short, then the vehicle is lined up to cross the bridge and can proceed to attempt to cross it. An example of how this is working is shown in figure 16, not that C is in a dummy location here for illustration.

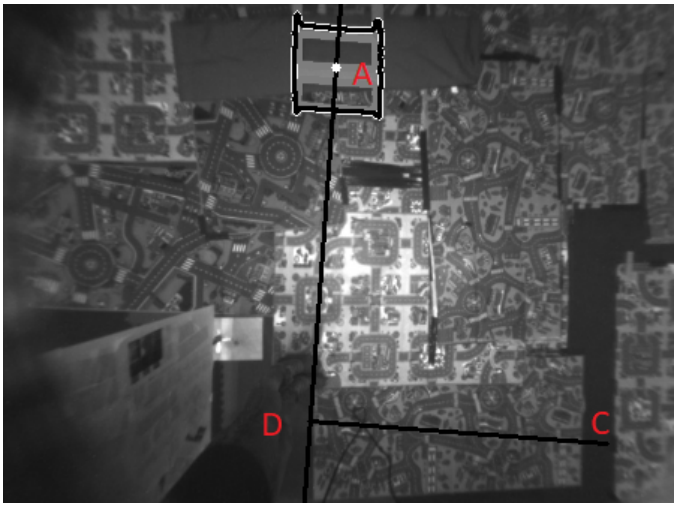

Figure 16: Example of calculations to approach bridge

VI. Land on the Bullseye

The bullseye target is placed very close to the bridge, close enough for the down facing duo camera to see it. In such a scenario, the bullseye looks fairly circular in the image and not an ellipse as it would, if the camera were observing it from a fairly large distance. Our basic approach was hence to find circles in the image.

We start off by doing some pre-processing on the image to extract regions of interest and reduce our search space for circle detection. The bullseye target is imprinted on a white glossy sheet. This feature allowed us to extract the region around the bullseye easily. We do the following:

1) Contrast limited adaptive histogram equalization to enhance the contrast to avoid overexposure on areas with a high intensity such as the target to loose information from there.
2) Median blur to remove noisy elements while preserving details on the edges.
3) Binary Theresholding to extract pixels with intensity greater than 80% of the maximum value.
4) Finding the largest contours and masking the resultant image with the original image.
5) Detecting Hough Circles on the resultant mask.

The output of the above mentioned steps are shown in Fig. 17. We noted that the same process worked well for the square target as shown in Fig. 18.

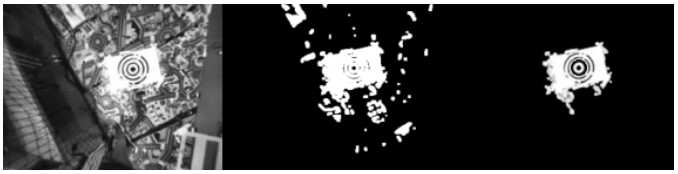

Figure 17: Bullseye processing

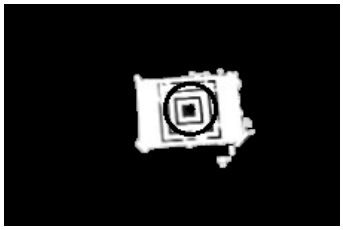

Figure 18: Hough circles applied on square target

VII. Yaw Troubles

Our Bebop had issues with yaw motions. We have tested our drone with a simple test movement script, and the same script with another drone and found a large difference in drift during a yaw manuever.

Our vehicle has been shown to drift by 1.5-2 meters for 90 degree yaw commands, and up to 3 meters for a 180 degree command. This amount of drift is completely unacceptable and very difficult to control. Resetting the firmware and other attempt at solutions did not fix the problem, as we found that the odometery origin point itself drifts. We were able to cut the drift down by aggresively countering the drift during a yaw manuever within the controller, but due to our dependence on odometry for this not all of the drift can be corrected.

This issue was solved by utilizing the down facing camera and modifying the script used to land on the bullseye. Because the code already exists for centering over the bullseye, and the bullseye dowed not move, we effectively have an real world anchor point to correct drift. Our strategy is to split the required yaw command up into several smaller pieces and correct the drift after each step by centering over the bullseye. This is an effective strategy but is quite slow, however due to the poor quality of our drone, this is the only way to succeed.

VIII. State Machine

The state machine is an overarching script that controls the spawn and despawn of the various nodes that are needed for this project. Each individual node knows its own condition for success as it performs some action to finish the task. For example, the window script knows when it needs to go through the window, the bridge script knows when it needs to cross the bridge, and the bullseye script knows when it lands. For each of these final task completing actions, a rospy shutdown command is issued afterward. This effectively kills the node.

The state machine simply calls these various nodes in order and waits until they die to call the next one. It is also responsible for small integration maneuvers between tasks. For example, the bridge cross requires quite a high altitude that is not necessary or beneficial for the bullseye. The state machine issues an altitude change command between those two nodes. Once the final node is complete, the statemachine has no more calls, and the race is finished.

IX. Results and Conclusion

In testing the obstacle course above, the algorithms were able to effectively navigate the course without issue. The

current implementation is robust, but slow and could use some optimization to improve speed. The majority of the speed improvements should be focused on image processing algorithms which would allow the controller to execute feedback more quickly.

X. Lessons Learned

- Expensive image processing operations cause a lag in image-based controller feedback operations which can cause erratic controller commands. Image processing requires a lot of computational power, and thus computation time. Complicated image operations cause a notable lag in image based feedback which is difficult to compensate for with a controller. All steps must be taken to reduce image processing time through reduced image size, optimal coding practices, and elimination of superfluous operations. Lag times in excess of 1 second are unacceptable and cause a notably poor controller performance. After optimization, we were able to reduce image lag to 0.2 - 0.6 seconds per frame. While not perfect, it resulted in a significantly better controller.
- LiPo batteries are sensitive to over discharge. Several LiPo batteries were harmed during the course of our testing. Damaged batteries can also have secondary consequences and inconsistent aircraft performance. For example, some batteries were more prone to allow the drone to drift during a yaw command.
- The hardware used during this course was not ideal. The Parrot Bebop drone is not consistent and exhibits some odd flight behaviors when given a command. We struggled to overcome issues in odometry and yaw drift where other teams did not encounter issue. Also, the wifi module is spotty and disconnects in flight. Other hardware issues such as broken duo and leopard camera connections also plagued the group. Overall, more robust hardware is desired for future work.

References

[1] Sanket, Nitin and Singh, Chahat; ENAE 788M: Hands On Autonomous Aerial Robotics; http://prg.cs.umd.edu/enae788m

[2] Sanket, Nitin; https://drive.google.com/file/d/1cAA2SxC-L51ifErkNYFfh_MINandCBce/view

[3] Open CV Tutorials: Contours; https://docs.opencv.org/trunk/d4/d73/tutorial_py_contours_begin.html

[4] Sanket, Nitin; https://drive.google.com/file/d/124j33YLm8Y6Hp-aauEZnCP2PheLix3i2/view

[5] Sanket, Nitin; https://drive.google.com/file/d/1jcbLtuw7ptxDDn0Djl9SUXXX8ENVdTu2/view

[6] Sanket, Nitin; https://cmsc426.github.io/gtsam/p4

[7] Andrew W. Fitzgibbon and Maurizio Pilu and Robert B. Fisher;Direct Least Squares Fitting of Ellipses, 1996 http://cseweb.ucsd.edu/ mdailey/Face-Coord/ellipse-specific-fitting.pdf