The Final Race

Mrinalgouda Patil
Alfred Gessow Center of Excellence
University of Maryland
College Park, Maryland 20742
Email: mpcsdspa@gmail.com

Curtis Merrill
Alfred Gessow Center of Excellence
University of Maryland
College Park, Maryland 20742
Email: curtism@umd.edu

Ravi Lumba
Alfred Gessow Center of Excellence
University of Maryland
College Park, Maryland 20742
Email: rlumba@umd.edu

***Abstract*—NEED ABSTRACT**

I. Introduction/Problem Statement

The goal of this project is to complete an obstacle course using the bebop parrot quadrotor. This obstacle course is composed of tasks that have been completed individually, so the major challenge is integration. In addition, a time limit of two minutes was imposed as an additional challenge.

This paper will first introduce the strategy for integration. Next, the methodology for each individual stage will be covered. This section also includes different approaches used to speed up individual sections. Finally, the results will be covered, including the final time breakdowns measured before and during the actual race.

II. Methodology

The general approach for this project was to start by solving each obstacle on its own. The starting point for each stage was the work done earlier in the semester, and from these baselines, certain refinements were made (using built in OpenCV features, updating controller tuning parameters, etc.). Each individual stage was tested for robustness on its own, and then deemed ready for integration. Note that optimizing performance was not done during this stage, as the focus was finishing the race before worrying about performance.

The strategy for integration was to use a modular approach. This involved writing individual codes for each obstacle and then combine them using a master code. This approach allowed for multiple people to work on different stages at once. During this stage, the approiximate position of each obstacle was used. For example, after finishing the window, the quad yaws by an approximate amount and moves relatively close to the bridge so that it can be seen. These ”searches” were performed instead of an actual search due to the time constraint on the mission.

After finishing integration, the entire code was tested for robustness. This involved perturbing different obstacles and testing the performance of the quad to ensure that the time constraint is still met. Finally, after robustness was achieved, the focus shifted to optimizing the performance. This involved identifying the stages that took the most time and looking for ways to reduce the time spent there.

III. Stages

This race involves 6 different stages, out of which 4 are unique. The first stage is a wall, specifically a low wall, that the quad must detect and fly over while staying between the two poles supporting the wall (Figure 1). Next, a yellow window must be detected and flown through (Figure 2). Next, the quad must cross the river, which it can only do over a bridge (Figure 7). The quad must identify the bridge and use it to cross the river. After the bridge, the quad must find a black and white circular bullseye and land on it (Figure 13). Next, the quad will take off and encounter a high wall (Figure 14). The quad must fly below the wall while remaining between the two poles supporting the wall. The final stage is to find and land on a black and white rectangular target (Figure 15).

The order of these stages is locked in, and will remain the same. The position of each obstacle will be known to the teams about an hour and a half before the live demo. Teams may come in and measure approximate positions during this time. Right before the demo commences, the professors will perturb all obstacles by small amounts in x, y, and ψ (yaw angle). The approach for each obstacle for each must be robust enough to handle the perturbations.

The following sections cover the detailed methods used for each obstacle.

A. Wall

The first wall is a low wall with features on it as seen in Figure 1.

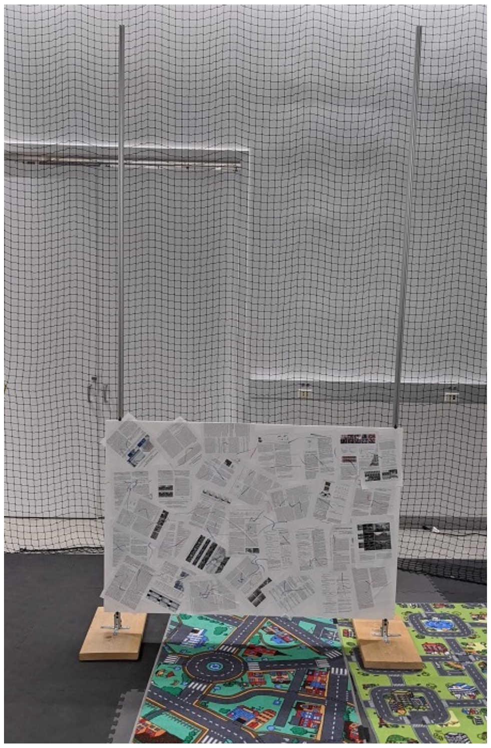

Fig. 1. The low wall has features on it to help with identification.

The strategy for this wall is identical to what was implemented in Project 4b earlier this year. However, certain shortcuts were implemented due to the fact that the initial position of the quadcoptor is given to the team and that we know this wall will be low. After takeoff, the quad quickly checks that it is not going to hit the pole. Then it moves up a half meter and flies forward past the wall.

The wall detection is done by identifying and matching features across two successive images. The locations of feature matches along with the disparities (how much the feature moved) are stored. A K means clustering algorithm is used to divide the feature matches up into foreground features (on the wall) and background features (not on the wall). Using the foreground features only, the center of the wall is obtained. Since we approximately know the depth to the window, we are just checking that the quad is pointing toward the center of the wall (roughly the center, really just checking if not going to hit the support poles). A more intensive check is done on the second wall, as the quad position relative to the wall is not known.

B. Window

After flying straight over the wall, the quad needs to fly through a yellow window (Figure 2).

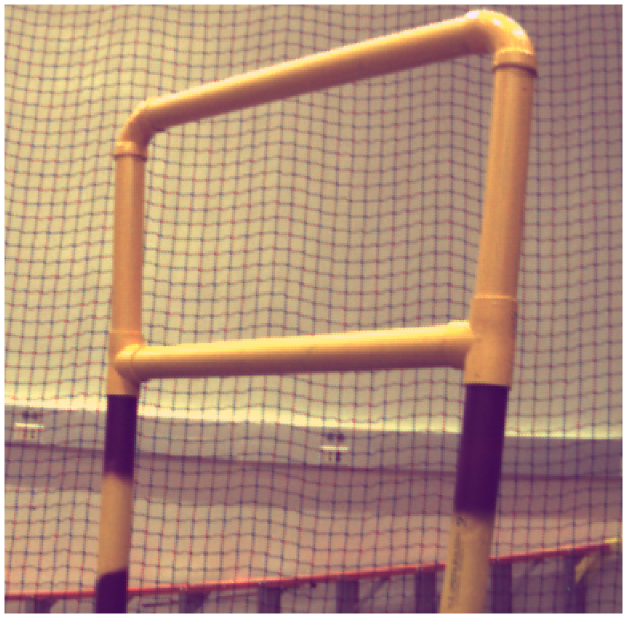

Fig. 2. The quad must fly through a yellow window.

The strategy for the window is similar to the approach used in Project 3a earlier this year with a few key changes. The general methodology is as follows: First, a gmm is used to create a mask with only the yellow from the image (Figure 3).

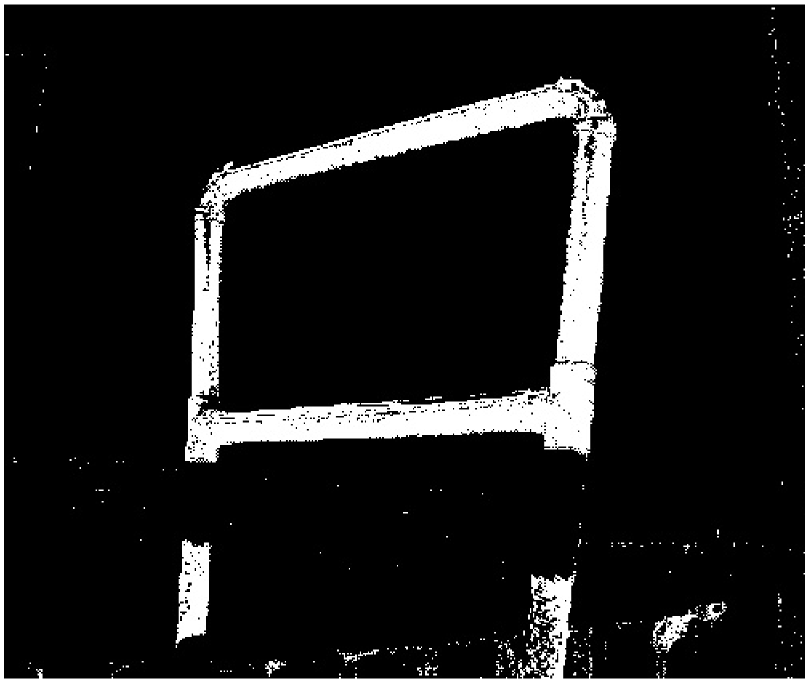

Fig. 3. A GMM was used to find the yellow gate.

Then, the edges are found using Canny Edge Detection. Next, Hough Lines are fit to the new mask (after Canny) and the intersections are computed. The intersections are grouped into four and represent the four corners of the window. These are given to PNP to solve for the window position relative to the quad. A waypoint roughly 1.2m back from the gate is set, and the quad follows the logic laid out in Figure 4.

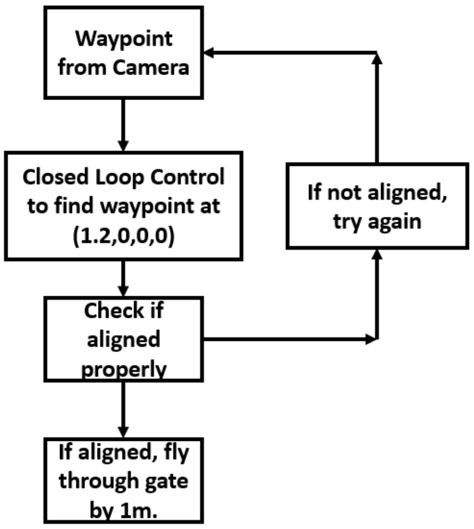
Waypoint from Camera
Closed Loop Control to find waypoint at (1.2,0,0,0)
If not aligned, try again
Check if aligned properly
If aligned, fly through gate by 1m.

Fig. 4. Controller Logic for window.

From last project, one major upgrade that was made is that a coordinate transformation was done to move the gate position from the camera frame to the inertial frame so that the proper waypoint in front of the camera would be found in the fewest iterations. Another major change was the convergence criteria. Before there were four criteria, one for each x, y, z, and yaw relative to the desired waypoint (with <10 degrees yaw). For this project, we realized that we don't have to necessarily fly through the gate perpendicular to it - we can line up offset in the lateral direction and fly through at an angle (as long as our yaw is in the proper direction (Figure 5). This is represented by having no y criteria and instead the following combined y and yaw criteria (for coordinate system, see Figure 5). However, despite the combined convergence check we kept an individual yaw criteria but significantly increased this value (25 degrees).

$$y(1 + sin(\psi)) < tol \tag{1}$$

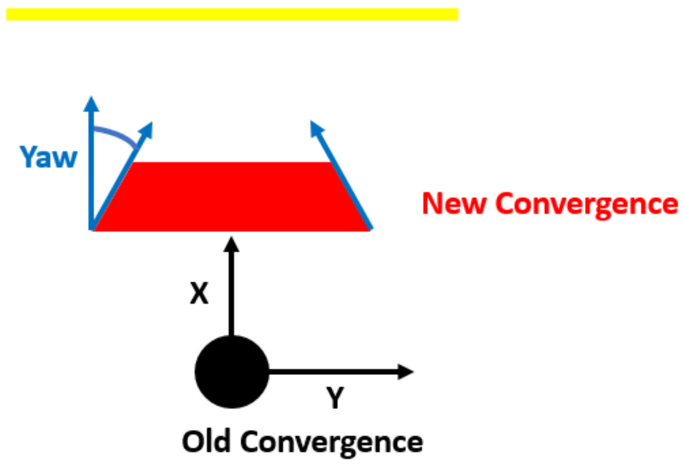

Fig. 5. New Convergence for Window Waypoint.

Finally, one of the main changes made between earlier in the semester and for this project was to change the resolution of the front camera from 640x800 to 480x640. This lead to a much higher field of view (from 25 to greater than 50 degrees). This allowed us to almost always see all four corners of the gate, instead of just being able to find two corners. This helps us reduce the time required for the gate, as there is relatively no search phase.

Looking at the lessons learned for that project, one of the main issues we had was that our PNP would give us errant messages. We addressed this with two separate approaches. First, we took multiple samples (3) and used this to calculate the quads position instead of relying on one sample. For every three samples, we would determine the standard deviation and throw out the grouping if the standard deviation was above a certain threshold. In addition, we would also throw out ”errant data”, such as a negative depth from the gate. Using this method helped give very good x, y, and z data, but the yaw data was still suspect.

Our yaw readings seemed to be the right sign, but sometimes the magnitude would be much higher than they should have been. To fix this, we only yaw 50 percent of the PNP-given yaw. This reduces yaw oscillations, while possibly adding one more iterations required for convergence.

After all of these changes, the window code became very robust, with a greater than 90 percent success rate. The time for the gate is relatively quick, taking as little as 20 seconds, and possibly up to 30 seconds. Although this is a large variation, this largely depends on how close to converged the initial position is, and we still almost always succeed in going through the gate.

C. Bridge

The strategy for the bridge detection changed the most from earlier in the year to the final project. The main reason for this is that the yaw angle of the quad relative to the river is arbitrary, and the code needs to be robust for any yaw angle. The method used for Project 4b relied on sweeping a box across the image and taking the standard deviation for a small subset of the image. If the standard deviation of this box was less than a certain threshold, it was determined to be the river. The largest gap in the river was determined to be the bridge (Figure 6).

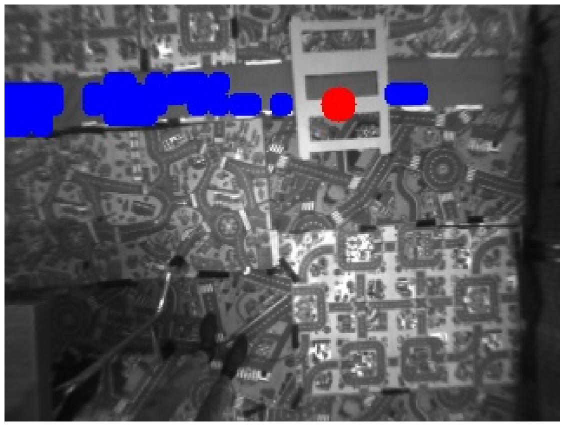

Fig. 6. The old method relied on finding the river using boxes. Then the largest gap in the river was the bridge.

However, this method only worked for small yaw angles (less than 10 degrees), as the boxes are moved across at constant yaw angles. Although this method could be adapted to different yaw angles, the computational time would increase significantly and the yaw angle would still have to be approximately known.

For this project, a new method was used that is much more robust. This procedure will be laid out for a case with roughly 30 degrees yaw (Figure 7).

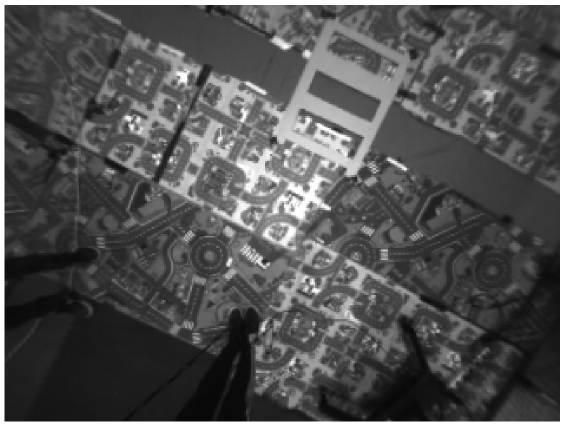

Fig. 7. The new method for the bridge can handle high yaw angles, such as what is present here.

First, Canny Edge Detection is used to identify all of the edges in the image (Figure 8). By looking at the image we see that both the bridge and the river have no edges (smooth). However, we also pick up the area without carpets and the concrete beyond the netted area as uniform - this will be addressed later.

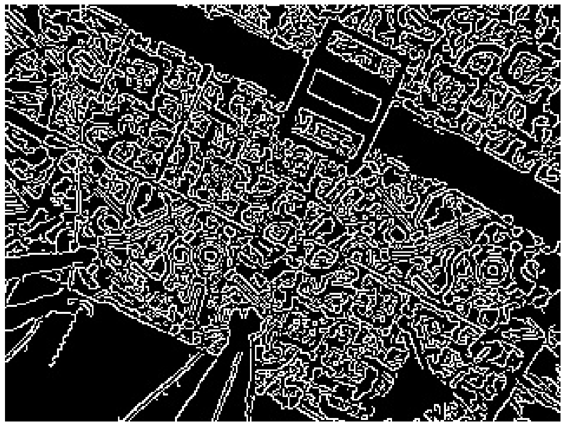

Fig. 8. Canny Edge Detection populates the noisy areas but leaves the uniform areas - including the river.

Next, dilation is performed to remove the noise of the carpet. Then, the image is inverted so that the uniform areas are white (Figure 9). In this mask we see that the bridge is mostly gone, but the river is still very clear. The effect of the black mats are still there - now we will remove those.

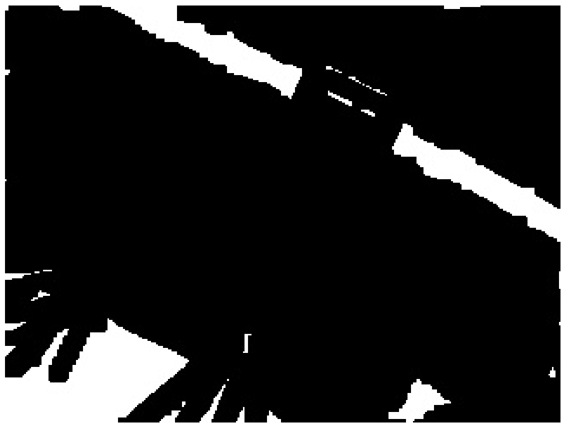

Fig. 9. Dilation helps fill in the noisy carpeted areas, leaving only the river and mat as uniform areas.

To remove the effect of the black mats, we create an additional mask where we threshold the image to remove the pixels below a certain value (the dark pixels). Notice that the black in this mask corresponds to the uniform dark mats.

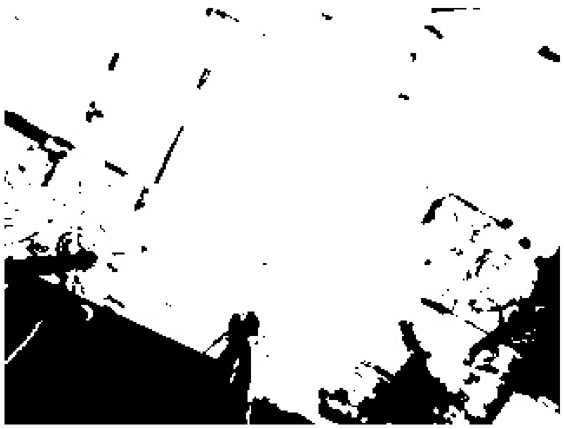

Fig. 10. If we threshold for dark, we see that only the matted surrounding areas show up - now we can remove them from our river mask.

When we combine the two masks together, we create a mask that only contains the river.

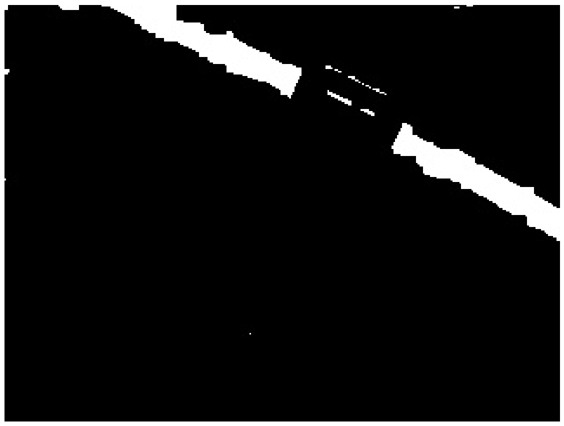

Fig. 11. By taking out the dark areas, only the river is kept.

Next, the river is identified using Hough Lines. The mask is normally clean enough to where all of the lines correspond to the river (they all differ by very small amounts). However, to ensure not outliers remain, a filter is applied. The median value of the angle θ and magnitude ρ that is used to describe the lines is taken and only lines that are within a certain ratio of the standard deviation of these lines is taken. During this process, care was taken to ensure that if ρ was negative the angle was flipped by 180 degrees and other similar transformations were made. Finally, based on this line, the yaw angle of the quad relative to the river can be found.

To find the bridge, the code checks every pixel along the line and looks for pixels that are above a certain value, which correspond to the brige. The average of all these image coordinates is taken, and the position of the center of the bridge in image coordinates is obtained. Based on the field of view of the camera and the altitude, we can calculate what the physical distance to this point is (assuming small angles for pitch and roll).

However, we want to move to a waypoint before the bridge so that we can use the bridge to cross the river. This waypoint is set as .5m behind the center of the bridge, perpendicular to the river. This position is calculated using the estimation to the center of the bridge and the yaw angle of the river. Figure 12 shows the river (blue), the center of the bridge (red), and the desired waypoint (green).

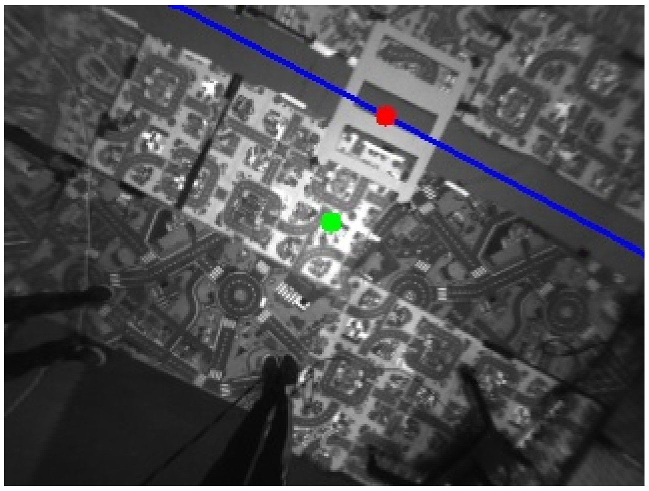

Fig. 12. Finally, this code gives the yaw angle of the river relative to the quad, as well as the desired waypoint (green) and river center (red)

Moving to this new method based on finding featureless areas using Canny Edge detection rather than sweeping a window made the code much more robust, especially for any yaw angle. This code has been tested up to 80 degrees initial yaw relative to the river, and works very well. The code is very consistent, running between 15 and 20 seconds every time.

D. Circular Bullseye

The methodology for the circular bullseye is very similar to what was used in Project 3b. The downward camera is used and the image is thresholded to only keep the very bright image (Figure 13).

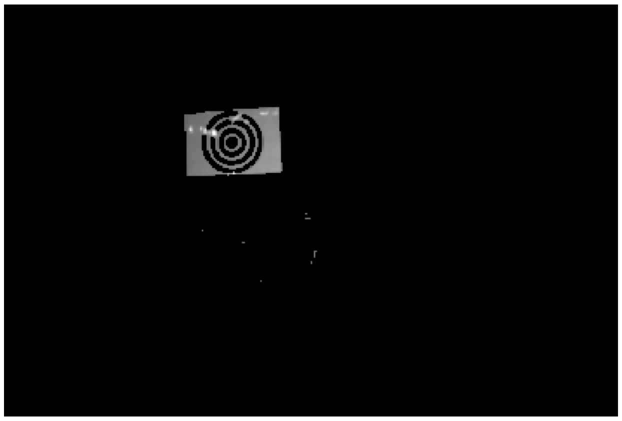

Fig. 13. Simple thresholding can isolate the target.

Next, Hough Circle is used to determine the center of the bullseye in the camera frame. Using this information, along with the FOV of the camera and current altitude (both known), the position of the target relative to the quad can be found.

E. Wall Again

The second wall is guaranteed to be a high wall (Figure 14).

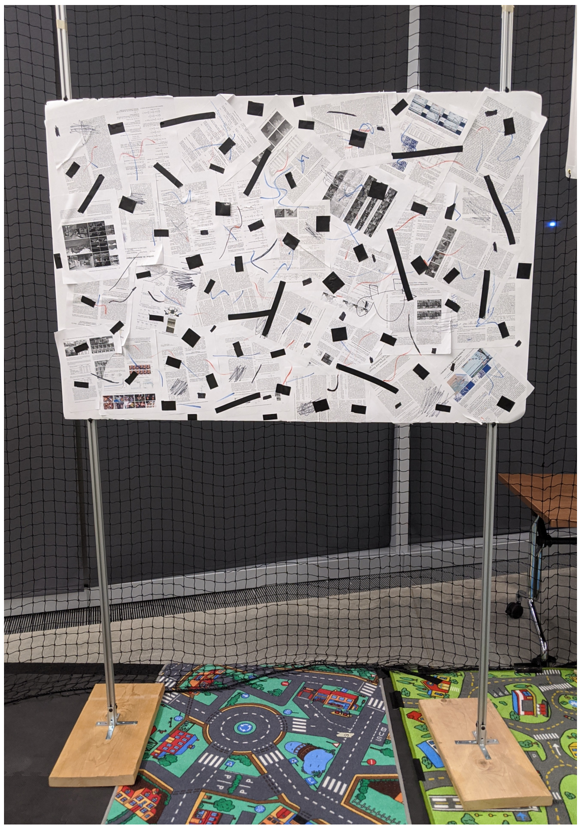

Fig. 14. The high wall was placed so the quad has plenty of space to fly underneath.

For this wall, two different approaches were taken. First, the perception approach actually determined the position of the wall and adjusted itself. This involved climbing to 1.5m so that the entire wall could be seen and then moving laterally until centered. Finally it would descend and fly through. However, moving up, running the perception code, and moving down again would take 5-10 seconds. As a faster alternative, a hard coded option was created. This allowed the distances (in x and y) that the quad would have to fly to get through the gate to be manually approximated and used instead.

F. Finish Tag

The last obstacle is a square bullseye (Figure 15). For this obstacle, the circular bullseye code was used. Originally, it was thought that Hough Rectangles would need to be used, however Hough Circles worked fine.

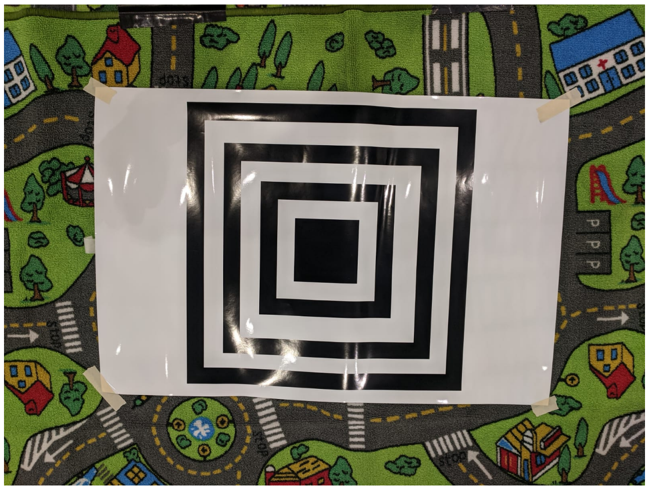

Fig. 15. The race is finished when the quad lands on the square bullseye

IV. Results

All of the stages were tested individually before integration. After integration, the total time of the race was determined, and based on this it was decided where to try to focus our energies to speedup. This is when we tried adding in hard coded options (for 2nd wall).

A. Race Results

On the day of the final demo, we had 3 successful runs. The first run used perception for the 2nd wall and took 95 seconds. The 2nd and 3rd runs used the hard-coded version of the 2nd wall code and the faster out of these was 82 seconds.

V. Conclusion and Lessons Learned

One major area that we could have improved on was our communication protocol. We ran our perception and controller codes at the same time - however only one was really working at once. The perception code would send a waypoint to the controller code, which would move to that location and then send a message to the perception code to run again (perception does not run while controller is running). This method ensured that our quad was relatively still while running the perception code, but it also took time to converge to one waypoint before moving to the next. Having the ability to update the waypoint during flight would reduce the time for convergence, especially for the window and the bridge.

Overall, we are happy with our approach and our results. The obstacle course was completed in less than the time limit, with time to spare. More importantly, the code was very robust, and the obstacle course would very consistently be completed in a similar way trial to trial.

REFERENCES

[1] ENAE788 Class 5 Slides
[2] Some Code taken from learnopencv.com/rotation-matrix-to-euler-angles/