ENAE788M Final Project - The Final Race

Estefany Carrillo, Mohamed Khalid M, and Sharan Nayak *

I. INTRODUCTION

This project involves providing the ability for a quadrotor to autonomously fly through an obstacle course comprising of a wall, a window, a bridge, a circular bullseye target, a wall again, and a square bullseye target. We use the front facing and down facing cameras as the sensors for detecting the obstacles and estimating the position of targets along the course. In this work, we have implemented techniques and algorithms developed in previous projects and combined them in an efficient way with the goal of completing the course in the shortest time possible. In the next sections, we describe the methods selected and modified in order to address each of the tasks during the course.

II. AVOID WALL

In this task, the quadrotor is given the task to fly over a wall on the floor. To accomplish this task, we first detect features in the wall using the function *detectandcompute* from the class *xfeatures2d* in Open CV [1], which returns the keypoints and descriptors of a given image in gray scale. Once features are obtained, we look for matches using the function *match* from the class *BFMatcher* available in Open CV between the features extracted from the current frame and a given template image of the wall.

To distinguish features of the wall from other features identified, we then use *DBSCAN* [2], a clustering technique available from *sklearn.cluster*. which is suited for discovering clusters with arbitrary shapes and for situations in which there is not enough information about the input parameters such as how many clusters to expect. The clustering algorithm works by computing the distance between every point and all other points and classifying the points based on the distance computed. Once the clustering technique is applied on the corresponding pixel locations of the matched features, we take the mean value of the pixel values corresponding to the largest cluster found.

To simplify the navigation control for flying over the wall, we specify a height for the quadrotor to reach after taking off and a distance for it to move forward once it moves up. We have tuned both of these distance values throughout our testing to ensure the quadrotor does not hit the wall and is well positioned for the next task. The benefit of using such control is that it is simple, fast and robust enough given that prior knowledge about the location of the wall is provided.

* The authors are affiliated with the A. James Clark School of Engineering, University of Maryland, College Park, 20742. Email:{ecarril2, khalid26, snayak18}@umd.edu

III. WINDOW DETECTION

For the gate detection, we use the Gaussian Mixture Model (GMM) segmentation [3] to segment the sides of the gate from the image (Fig. 1). We then input the image to an image processing pipeline consisting of dilation, erosion and connected component labeling to correctly extract out the gate from the background. We then determine the centroid of the largest connected component, corresponding to the gate as shown in Fig. 2, to determine the centroid of the gate.

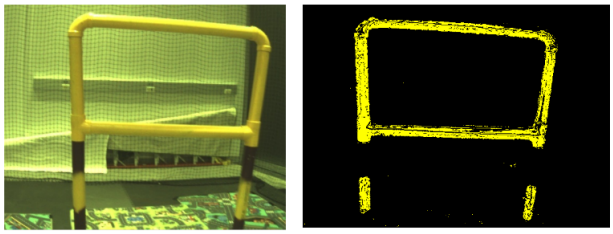

Fig. 1: Original and segmented image using GMM

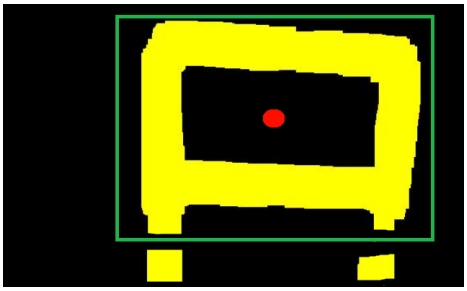

Fig. 2: Centroid of the largest connected component

After flying over the wall and moving forward, the quadrotor is commanded to move in the Y-direction by a fixed amount and yaw by a fixed amount in order to orient the field of view of the front camera towards the direction of the window. Once the quadrotor yaws, it proceeds to perform the window detection and continues to align with the window's centroid.

IV. BRIDGE DETECTION

We use the down facing camera to perform detection of the bridge. We first take the image of the bridge (offline) and use that as the template for the detection of the bridge. The features of the bridge are determined using Scale Invariant Feature Transform (SIFT) [4] after power-on of the drone

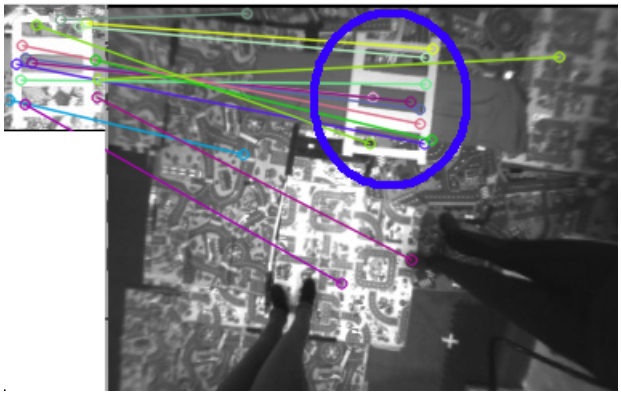

Fig. 3: Template Bridge Image (Left) and Test (Right) Image for Bridge Detection. The features in the blue oval corresponds to the features in the bridge and the remaining features correspond to outliers which are removed.

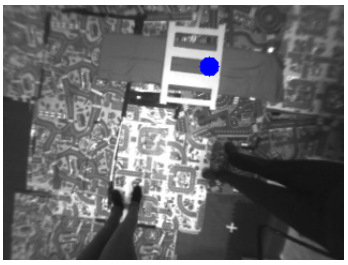

Fig. 4: Centroid of detected bridge which corresponds to the mean of all identified feature locations.

and cached for later use. We then take images from the down facing camera and detect features in the image using SIFT. We then match the features of camera image and the saved template bridge to find the bridge in the image. During the matching, there are some features that are outliers which do not correspond to features in the bridge. We use DBSCAN algorithm to remove outliers that do not corresponds to bridge features. We then take the mean of the locations of all features to get the centroid of the bridge. We observe that the centroid calculated does not always correspond to the true centroid of the bridge but based on our tests, we found this method to be accurate enough for us to detect and cross the bridge. The advantage of using SIFT is that it is scale invariant and works for drone flying at relatively range of different altitudes. The outputs of the feature matching and bridge centroid detection are shown in Fig. 3 and Fig. 4, respectively.

After going through the gate, the quadrotor proceeds to acquiring images from the down-facing camera and running the bridge detection algorithm in order to align with its center and fly over it.

V. CIRCULAR BULLSEYE DETECTION

For the circular bullseye detection, we use the Gaussian Mixture Model (GMM) segmentation to segment circular bullseye from the background. (Fig. 1). We then apply Hough circles transform to detect the circles or ellipses in the bullseye. The center of the detected circles correspond to the center of bullseye.

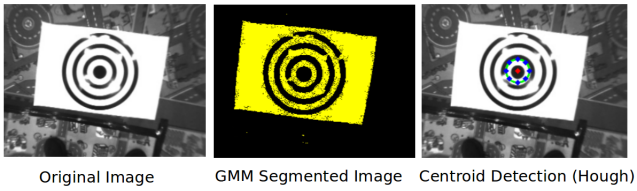

Fig. 5: Bullseye centroid detection

After flying over the bridge, the quadrotor proceeds to acquiring images from the down-facing camera and running the bullseye detection algorithm in order to align with its center. Once the quadrotor finishes aligning, it is commanded to land. To finish the last part of the race, the quadrotor is commanded to yaw by a fixed amount in order to face the second wall. Then, it is commanded to take off and move forwards for a specified distance to then proceed to perform the square bullseye target detection described next. This is done to make sure the bullseye target is searched for after passing through the wall.

VI. SQUARE BULLSEYE DETECTION

For the square bullseye detection, we use the Gaussian Mixture Model (GMM) segmentation to segment the bullseye from the background and train the GMM model to detect white segments. The segmentation output from the GMM segmentation is shown in Fig. 6. We then apply morphological operations on the output image by eroding it using a kernel defined by a 3×3 matrix of ones. This helps remove the noise detected outside the bullseye target. Then, we perform dilation to enlarge the segmented areas and fill in the small gaps in the bullseye segment. We do this by using a kernel defined by a 15×15 matrix of ones. Lastly, we perform erosion again to ensure that noise generated by the dilation is reduced. The output image of the morphological operations is shown in Fig. 7.

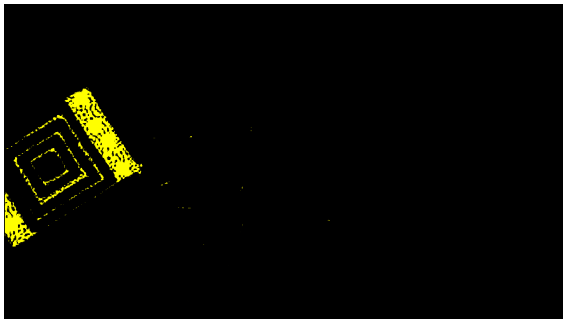

Fig. 6: Square bullseye image GMM segmentation output.

The next step is to detect the largest connected component in the image using the function *connectedComponentsWithStats* available in Open CV. This function will output all the connected components identified in the image in binary scale. We then compute the percentage area of each component

Fig. 7: Output from morphological operations performed on segmented image.

and select the component with the largest percentage area. If this percentage area is larger than a user-defined threshold, then we consider this component as a candidate. All pixel values not corresponding to the candidate component are set to 0 and the rest to 1. Hence, we obtain an output image in binary, which is converted to grayscale. Then we perform Canny-edge detection on the output image and apply thresholding on the image using the function *threshold* with *THRESHBINARY* type from Open CV. Once we obtain the resulting output, we look for contours in the image by applying the function *findContours*. And since we are only interested in closed contours, we add only the convex contours to the list of possible candidates. We then sort the candidates by their area and compute the center of each detected contour in order to generate a combined center. The identified largest connected component is shown in Fig. 8. The last step is to calculate the center of the closed contours by using the function *moments* in Open CV. The identified center is shown in Fig. 9.

Fig. 8: Largest connected component identified in the square bullseye image.

After passing through the wall, the quadrotor runs the detection method described and aligns with the bullseye centroid prior to landing, at which point it terminates its mission.

In order to determine when the different methods described above need to be invoked and run while minimizing the utilization of computational resources, we have designed a software architecture to handle this logic as described in the next section.

Fig. 9: Square bullseye centroid detection.

VII. SOFTWARE ARCHITECTURE

We define a software architecture to guide the quadrotor's navigation through the course. The architecture consists of a state machine, used to determine the next task the quadrotor needs to perform based on the current task it is performing. Thus, the state machine keeps track of the state of the quadrotor. There are 7 states defined, one for each task in the course:

$$STATES = \left\{ \begin{array}{l} \texttt{LAUNCH} \\ \texttt{CROSS FIRST WALL} \\ \texttt{CROSS WINDOW} \\ \texttt{CROSS BRIDGE} \\ \texttt{LAND CIRCLE BULLSEYE} \\ \texttt{CROSS SECOND WALL} \\ \texttt{LAND SQUARE BULLSEYE} \\ \texttt{FINISH} \end{array} \right\} \quad (1)$$

The initial state is `LAUNCH` and once the quadrotor starts its mission, it executes the control corresponding to this initial state. Once the quadrotor finishes the action of taking off, it sleeps for a few seconds and moves on to its next state (i.e. `CROSS FIRST WALL`). At each state, a detector class and its corresponding controller class are invoked to first identify the obstacle or target specific to the task and control the navigation of the quadrotor in order to avoid it or reach it, respectively. Only when detecting the specific objects, the front camera or stereo camera inputs are accessed. The front camera images are used to detect the walls and the window while the stereo camera images are used to detect the bridge and bullseye targets. Furthermore, we perform tuning of the controller gains to adjust velocity commands in order to ensure the quadrotor completes the course successfully. We use a single ROS node to run the state machine. All controllers and detector are implemented as separate classes. The front camera and stereo camera acquisitors are run on its own separate threads. A diagram of the overall architecture demonstrating how the state machine, detector and controller classes, and inputs from the cameras are interconnected is shown in Fig. 10.

VIII. MOTOR CONTROLLER

For navigation control, we have opted for an open loop motor controller tuned to specific distances between obstacles to move from one obstacle to another. We use a

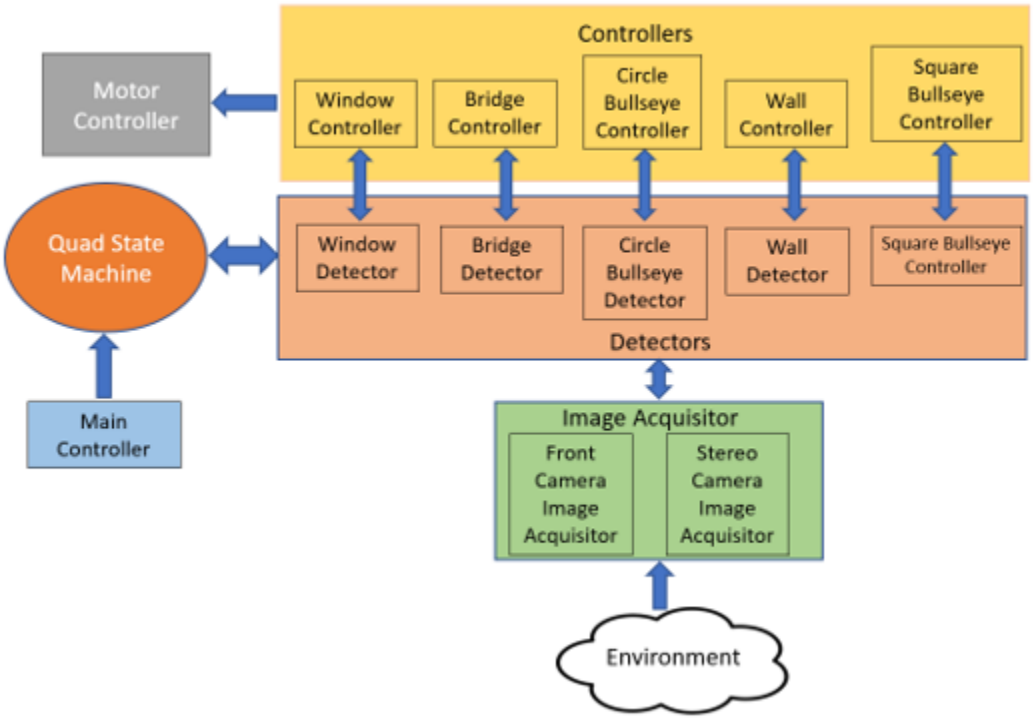

Fig. 10: Software architecture used in the implementation of the quadrotor's navigation through the course.

proportional controller with saturation limits to align with the window, bridge and bullseyes. The saturation limits cause a jerkiness in the movement of the quadrotor but it helps prevent rapid change of quadrotor's location during control which leads to more stability. We also use the odometry information to perform a yaw correction after take-off from the circular bullseye. The yaw correction is calculated using the initial yaw during takeoff and the final yaw just before landing on the circular bullseye. We then use the prior yaw calculated between the initial and desired final heading of quadrotor (approximately 200°) to align with the square bullseye.

IX. CONCLUSION AND FUTURE WORK

This project involved flying the quadrotor through a race course. The front camera, downward facing stereo camera and odometry estimates were used to guide the quadrotor along the course. We use image processing techniques like GMM color segmentation, determination of largest connected component, Hough lines and circles, Feature detection and matching to detect the various obstacles in the course. We use prior measured pose of the obstacles to initially move and align with the obstacles.

Although we are able to complete the course using our current image processing techniques, there is room for improvement. First, we would like to use PnP to align and move towards the gate. Our current PnP estimation suffers from noise and we would like to use better filtering techniques to improve PnP estimate. Secondly, we use strong prior information about the positions of the wall to avoid hitting the walls rather than detecting and avoiding it. Our future work will include calculating relative depth estimates of the wall and background to avoid hitting the wall. Thirdly, our controller suffers from jerkiness of movement during control. We would like to use more advanced controllers like PID or Model Predictive Control (MPC) to get smooth trajectories of the quadrotor. Finally, we would like to make our image processing pipeline more time-efficient and streamlined, and consider using deep-learning based image processing to improve speed.

X. ACKNOWLEDGEMENT

The authors feel they have learned a lot from the course and would like to thank the Aerospace Engineering and Computer Science department for making this course happen. We especially like to thank the instructors Nitin J. Sanket and Chahat Deep Singh for designing this course and teaching us various concepts in Computer Vision, and helping to debug the numerous issues with the drone and promptly answering all questions related to the different projects.

REFERENCES

[1] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media, Inc., 2nd edition, 2013.

[2] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[3] Jong-Hyun Park, Wan-Hyun Cho, and Soon-Young Park. Color image segmentation using a gaussian mixture model and a mean field annealing em algorithm. *IEICE TRANSACTIONS on Information and Systems*, 86(10):2240–2248, 2003.

[4] David G Lowe et al. Object recognition from local scale-invariant features. In *iccv*, volume 99, pages 1150–1157, 1999.