P5:The Final Race!

Shakthibala Sivagami Balamurugan
Robotics Engineering
Worcester Polytechnic Institute
Email: sbalamurugan@wpi.edu

Aditya Patwardhan
Robotics Engineering
Worcester Polytechnic Institute
Email: apatwardhan@wpi.edu

Harsh Shah
Robotics Engineering
Worcester Polytechnic Institute
Email: hshah2@wpi.edu

***Abstract*—The aim of the project is to demonstrate a vision-based approach for autonomous drone navigation through obstacles of known windows, unknown windows and returning back through them. We developed a perception and navigation stack for autonomous navigation in Vizflyt using Unet for semantic segmentation and optical flow using RAFT. We use these estimates and classical computer vision techniques for gap detection followed by visual servoing and navigation of the drone through the gap.**

I. Introduction

The capability of navigating a complex environments is challenge for autonomous drones. We aim to enable the drone to autonomously identify and navigate through gaps of unknown shape, size and location. The process is initiated with the drone flying to position where the gap is visible in its frame. We detect the gap using optical flow from Recurrent All-Pairs Field Transforms (RAFT) network. We maneuver the drone horizontally and vertically after a gap has been detected. This movement cause the foreground and background (gap) to have different magnitudes of optical flow. We use this to get the regions of gap in the textured wall. In the next phase we process this flow and use canny edge detection to get the gaps outlines. We then find the largest contour and calculate its centroid using moments. We now have the center of the unknown gap, we need to center the drone such that its image center is aligned with the center of the gap. By using Visual Servoing we are able to accomplish centering the drone with the center of the gap. We then go through the gap, achieving the goal of autonomous navigation through unknown gaps. We had divided the obstacle course broadly into 3 stages. Each of the stages has a different set of challenges that need to be tackled for functioning.

Fig. 1. FPV view of drone in vizflyt

II. Navigating Racing Windows

For Navigation through the racing windows, we adopted a U-Net architecture, a fully convolutional encoder–decoder network designed for dense prediction tasks. This architecture had previously been used developed and validated as part of an earlier project, and was able to detect the windows with dice scores of 94.27% and IoU scores of about 89.21%.

Initial experiments with the standard U-Net architecture yielded high Dice scores but failed to properly segment window corners—a critical requirement for accurate pose estimation. The network struggled particularly with low-contrast backgrounds where window edges were subtle. Additionally, the model produced multiple detections on the same frame. This required a post-processing step to filter these detections and obtain a single, robust mask for the window.

A. Data generation

.The data generation section contains two section, One is Dataset generation using blender and the other is Neural Network using Unet

1) : sectionDataset Generation using Blender The Window template is given from which the dataset generation has to begin. To train our neural net model for window segmentation we had to build our dataset containing images and labels. The images used in the dataset is trained on 3D blender environment. A multitude of scenarios were considered with varying camera angles, lighting conditions, occlusions and number of windows. Then the dataset created was augmented with varying camera placements and lighting conditions. Pictures of Laboratory kind of setup were chosen to make the model robust to handle any kind of scenarios for detection.

Initially, We did on random backgrounds which affected our model accuracy, The windows shown acted as out-of-distribution dataset from the drone fpv view, Hence to address this problem, We chose laboratory with netted setup which was similar background as in vizflyt, With this small change and increase the dataset size, Our model could perform way better than previously it was.

2) Dataset Characteristics: The final dataset used for training had the following properties:

- **Total images:** 12000
- **Image resolution:** 640×640 pixels
- **Format:** Paired RGB images and binary segmentation masks

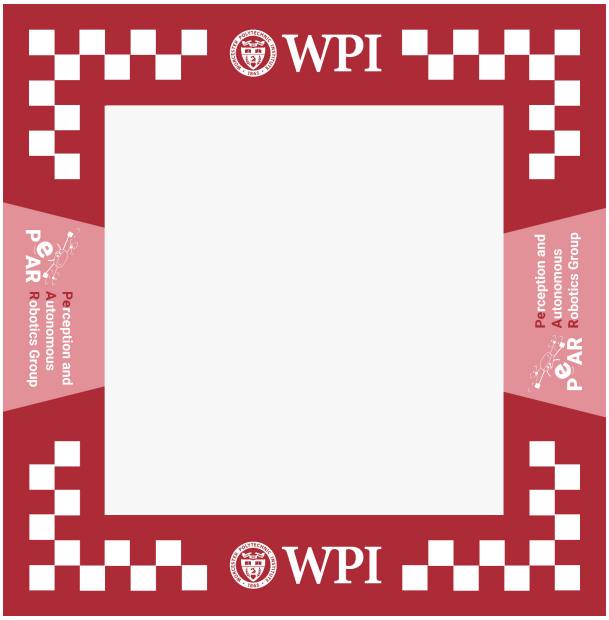

Fig. 2. Window Template

Fig. 3. Domain Randomization with the template

- **Theme:** Laboratory-style backgrounds for domain alignment

3) Data Augmentation Pipeline: To ensure the model's robustness and its ability to generalize to diverse, real-world conditions, we implemented an extensive augmentation pipeline using the Albumentations library.

These augmentations were applied only during the training phase. The validation and test sets used only resizing and normalization to accurately evaluate the model's true performance.

4) Augmentation Techniques: The pipeline included a combination of spatial, color, and noise-based augmentations:

- **Spatial Augmentations:**
  - Horizontal flip ($p = 0.5$)
  - Vertical flip ($p = 0.5$)
  - Random rotation ($\pm 45°$, $p = 0.5$)
- **Color Augmentations:**
  - Grayscale conversion ($p = 0.1$)
  - Random brightness and contrast adjustment ($p = 0.3$)
- **Noise and Blur Augmentations:**
  - Gaussian noise ($p = 0.4$)
  - Gaussian blur with kernel size 3–7 ($p = 0.4$)
  - Motion blur with kernel size 7 ($p = 0.3$)
  - Sharpening with $\alpha \in [0.2, 0.5]$ ($p = 0.4$)
- **Dropout Augmentation:**
  - Coarse dropout with 3–8 holes ($p = 0.4$)
  - Hole dimensions: 5–20 pixels
- **Normalization:**
  - Normalization using ImageNet statistics:
    - mean=$[0.485, 0.456, 0.406]$
    - std=$[0.229, 0.224, 0.225]$

5) Neural Network using Unet: For the segmentation component, we adopted the U-Net architecture, a fully convolutional encoder–decoder network designed for dense prediction tasks. The encoder path captures progressively abstract feature representations through convolution and pooling, while the decoder reconstructs the spatial resolution using transposed convolutions. Skip connections between mirrored levels of the encoder and decoder facilitate the fusion of contextual and spatial information, enabling precise delineation of structure boundaries. Given the task-specific nature of our dataset, the model was implemented in PyTorch and trained from scratch using custom dataloaders.

6) Initial Challenges and Post-processing: Initial experiments with the standard U-Net architecture yielded high Dice scores but failed to properly segment window corners—a critical requirement for accurate pose estimation. The network struggled particularly with low-contrast backgrounds where window edges were subtle.

Additionally, the model produced multiple detections on the same frame. This required a post-processing step to filter these detections and obtain a single, robust mask for the window.

7) Boundary Loss Function: To improve corner detection, we introduced a boundary-aware loss function that heavily penalizes prediction errors at window edges. This formulation ensures the network learns sharp, well-defined boundaries while maintaining global shape accuracy.

The boundary loss computes edge maps using Sobel operators and applies an increased penalty ($5\times$) at the boundary pixels. The process is as follows:

- Compute horizontal (G_x) and vertical (G_y) gradients using Sobel kernels.
- Calculate the edge magnitude: edges $= \sqrt{G_x^2 + G_y^2}$.

- Apply a binary threshold to the edge map (e.g., threshold = 0.5).
- Weight the Binary Cross-Entropy (BCE) loss: $\text{loss} = \text{BCE}(\text{pred}, \text{target}) \times (1 + 5 \times \text{edges})$

8) Attention U-Net Architecture: To further improve performance on low-contrast backgrounds, we integrated attention modules into the skip connections of the U-Net. The attention mechanism helps the network focus on relevant spatial regions during feature concatenation from the encoder to the decoder path.

Each attention block takes two inputs:

- A gating signal (g) from the decoder path.
- Skip connection features (x) from the encoder path.

This allows the model to learn to suppress irrelevant background regions while amplifying features relevant to the window segmentation task.

We trained the model on dataset of 12k images, for 10 epochs with a batch size of 4 . We got Test Dice Score: 0.94 and Test IoU Score: 0.89

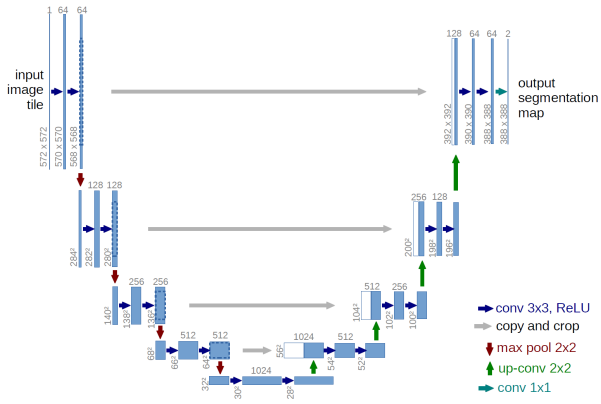

Fig. 4. Unet architecture

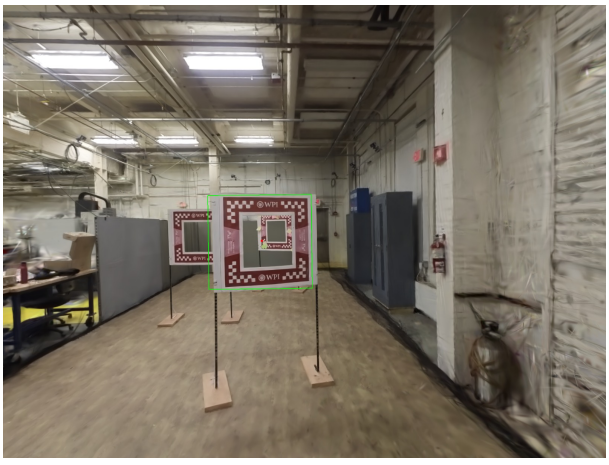

Fig. 5. Detected Window by Unet

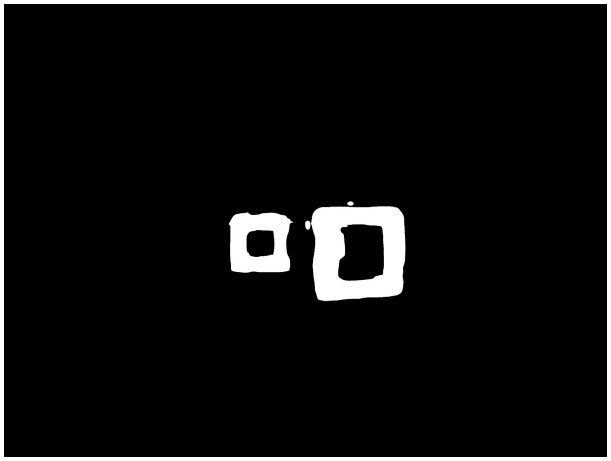

Fig. 6. Model's Output mask

B. Algorithm for Waypoint Navigation using Monocular camera

The following algorithm describes the autonomous navigation strategy used to sequentially traverse multiple windows using only monocular vision:

- The segmentation model is executed on the onboard CPU at 5 Hz to generate a post-processed binary mask of the detected window at each iteration.
- Upon obtaining the mask (every 0.2 seconds), the pixel displacement between the detected window center and the image center is computed in the x and y axes. The drone is then commanded to minimize this error through visual servoing for lateral and vertical alignment.
- Experimental trials demonstrated that approximately six iterations are sufficient for the drone to converge to the window center. A fixed error threshold was not used, as the positional deviation varied across runs and did not consistently fall within a predefined tolerance band.
- Once alignment is achieved, the drone proceeds forward along the x–axis. The detected window area in the image frame serves as a proxy for distance: as the drone advances, the window occupies a larger pixel area. Based on empirical observations, a forward motion duration of three seconds was found to be sufficient to traverse the window.
- This approach enabled successful traversal of the first two windows. However, the third window initially posed a challenge due to being outside the camera's field of view.
- To address this, a yaw-search behavior was integrated. When no window is detected, the drone executes a yaw motion to scan the environment until a window reappears in the camera frame.
- With the inclusion of the yaw-based search mechanism, the drone successfully navigated through all three windows autonomously.

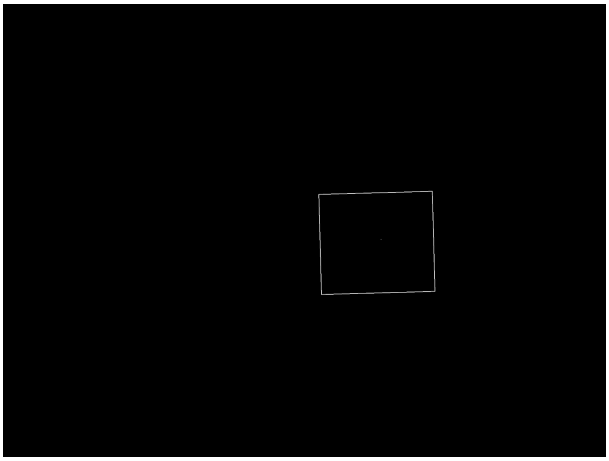

Fig. 7. Post processed mask

III. Navigating the Unknown Window

To enable autonomous traversal through an unknown window, our approach integrates robust perception and control methods that allow the quadrotor to detect, localize, and align itself with the gap in real time. The pipeline begins with evaluating multiple optical-flow–based techniques for reliably identifying the window opening under challenging visual conditions, followed by a post-processing stage designed to extract a stable centroid of the gap. This visual information is then incorporated into a lightweight proportional visual-servoing controller that iteratively drives the drone toward alignment with the window center. The following subsections detail the progression of methods explored, the reasoning behind the final selected approach, and the control strategy used for precise navigation.

A. Optical Flow using Farneback method

We first implemented Farneback method for optical flow using opencv library. This method was used to differentiate foreground and background using thresholding on the flow output. While this method was fast it did not provide optical flow. Even with post-processing with magnitude filter, canny edge detection and contouring we were not able to get good results for gap centroid.

B. Window Finder

Our drone camera field of vision is wide due to which the textured wall is a small part of the image at start. This results in a lot of the background having similar properties as the gap in the textured wall. This led to improper detection which was difficult to solve through post processing. We tried to segment the textured window using classical cv methods like canny and contouring. Due to the texture of the wall this led to subpar window boundary detections and the brick texture overloaded the contouring methods. We also realized that not having a consistent crop would break the optical flow, so we did not use this approach.

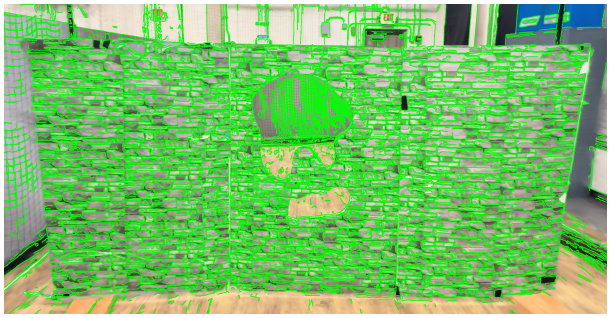

Fig. 8. Canny edge detection for preliminary window finding

C. Optical Flow using Dense Inverse Search

While implementing the Dense Inverse Search (DIS) Algorithm to compute optical flow, which is also a function of the opencv library, the results were not that promising. Even though the model was generating better results than Farneback, these results were not yet near usable for gap centroid processing.

D. Optical Flow using RAFT

The Recurrent All-Pairs Field Transforms (RAFT) Algorithm for optical flow is known to achieve high accuracy with method essentially combining the correlations with a recurring refining mechanism. While testing the available images with the pretrained raft models, crisp images were generated that detected the window gap and the centroid with great precision. While testing the models on different pretrained RAFT models, we were able to determine that different raft models were actually better for detection in images and in videos. The model trained on the *Things3D* dataset was detecting better for still images, while the model trained on the *Sintel* dataset was detecting the gap window with a very good accuracy.

E. Post-Processing

Following the optical flow estimation, we implemented a magnitude-based segmentation pipeline to isolate the target gap. We operated on the premise that during forward flight, the gap (representing the distant background) exhibits significantly lower optical flow magnitude compared to the window boundaries (foreground).

First, we applied an inverse binary threshold to the flow magnitude matrix. By setting a magnitude threshold of $\tau_{mag} = 2.0$, we generated a binary mask where pixels exhibiting flow below this value were set to high (255), effectively highlighting the static or distant regions representing the gap.

To ensure spatial coherence and reduce noise artifacts, we employed a two-stage morphological filtering process. A morphological *close* operation was first applied to fuse fragmented regions and fill small holes within the detected gap area. This was immediately followed by a morphological *open* operation to eliminate small, stochastic noise speckles from the surrounding regions, resulting in a clean binary segmentation of the low-flow areas.

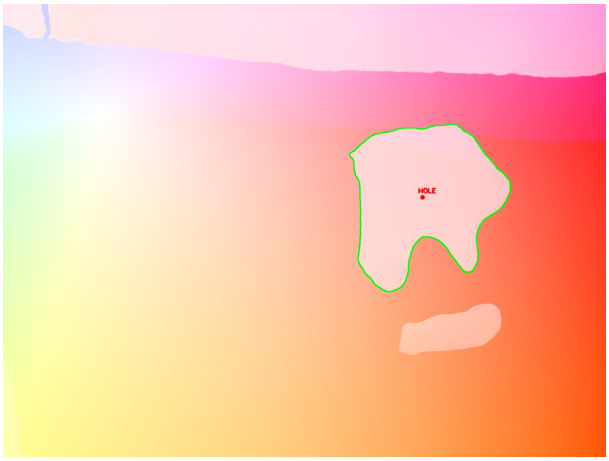

Fig. 9. RAFT with gap centroid

Fig. 10. Frame while coming back

From this processed mask, we extracted contours to delineate the boundaries of potential gaps. We filtered these candidates by area, selecting the largest contour as the primary target. Finally, we computed the first-order image moments of this largest contour to calculate its centroid (C_x, C_y). This coordinate pair serves as the visual servoing setpoint, allowing the quadrotor to align itself with the center of the gap for safe traversal.

IV. Returning to the Start

The final stage of the course required the aerial robot to return from the end point to the origin as quickly as possible while ensuring collision-free navigation. During this return leg, the vehicle must traverse the same sequence of openings—first the unknown window and subsequently the racing windows—but approached from the reverse direction. A key challenge arises from the fact that all windows appear uniformly white when viewed from behind, preventing the use of visual cues or feature-based perception for real-time localization of window boundaries. To address this, we implemented a waypoint recording and following pipeline for the course.

A. Forward-Pass Waypoint Logging

During the forward traversal, the vehicle's estimated pose (x,y,z,ψ) is continuously recorded and appended to a CSV file at a fixed frequency. These waypoints capture the spatial trajectory followed by the drone on its forward navigation leg. The recorded path inherently reflects all necessary corridor alignments, window centerlines, and dynamic adjustments made by the controller or perception system.

B. Return-Pass Waypoint Tracking

Once the vehicle reaches the end point, the recorded CSV file contains a complete ordered list of outbound waypoints. For the return phase, this list is reversed and used as the reference trajectory. The return-flight controller operates in waypoint-tracking mode, sequentially commanding the drone to navigate toward each reversed waypoint. A smooth trajectory is generated using interpolation between waypoints or by applying a position-tracking controller (we tested both PID and geometric controller) . This enables the drone to retrace the same corridor with consistent speed while ensuring stability.

Handling Back-Facing (White) Windows

Since windows do not provide fiducial markers or texture when viewed from behind, conventional perception modules will fail to detect edges or openings reliably. Instead of attempting back-facing window detection, our method circumvents the problem entirely by ensuring the drone is already aligned with windows due to the stored trajectory.

Termination and Time Recording

The run is considered complete when the drone reaches the origin within a specified tolerance or when the controller violates safety constraints leading to termination. In both cases, the elapsed simulation time is logged as the official result. Because the return trajectory is deterministic and follows a proven corridor, this method minimizes the risk of collision-induced termination and allows for reproducible timing across multiple trials.

V. Controller

A. PID

We employed PID control for P3 and P4, tuned for a 0.3 splat-scale velocity. However, during the P5 drone race, we observed that PID control led to unacceptable settling times, which negatively impacted overall performance. To avoid compromising on time, we opted for a geometric controller. While PID performs well for smooth trajectories, it is less effective during aggressive and complex maneuvers.

B. Geometric Controller

Due to the stability issues in PID at the start, our team tried implementing a geometric controller for quadrotor trajectory tracking using the SE(3) formulation. This approach helped us directly regulate the position and attitude in SO(3), ensuring

TABLE I
PID GAINS FOR POSITION AND VELOCITY CONTROLLERS

Controller	P	I	D
x-position	0.725	0.00	0.68
y-position	0.20	0.00	0.78
z-position	2.30	0.01	3.00
x-velocity	2.80	0.20	0.30
y-velocity	0.70	0.10	0.10
z-velocity	15.00	4.00	0.10

smooth behavior under large rotations and avoiding singularities. We integrated our integration directly into our simulation pipeline for real-time control and ease of running.

1) Position Control: Given the desired position x_d, velocity v_d, and feed-forward acceleration a_d, we compute the commanded acceleration

$$a_c = a_d + K_p(x_d - x) + K_d(v_d - v), \tag{1}$$

which is then combined with gravity to obtain the desired force

$$f_{\text{des}} = a_c + ge_3. \tag{2}$$

The normalized vector

$$b_{3_d} = \frac{f_{\text{des}}}{\|f_{\text{des}}\|} \tag{3}$$

defines the desired body z-axis and links position error to vehicle orientation.

2) Desired Attitude and SO(3) Error: A desired yaw ψ_d provides a reference heading direction. Using cross products, we construct the desired attitude matrix

$$R_d = \begin{bmatrix} b_{1_d} & b_{2_d} & b_{3_d} \end{bmatrix}. \tag{4}$$

The geometric attitude error is computed on $SO(3)$ as

$$e_R = \frac{1}{2} \vee \left(R_d^\top R - R^\top R_d \right), \tag{5}$$

and the desired body-rate command is

$$\omega_d = K_R \, e_R. \tag{6}$$

TABLE II
GEOMETRIC CONTROLLER GAINS FOR POSITION, VELOCITY, AND ATTITUDE

Axis	$K_{p,\text{pos}}$	$K_{d,\text{vel}}$	$K_{p,\text{att}}$
x	0.15555	0.15555	0.15555
y	0.15555	0.15555	0.15555
z	0.00080	0.00080	0.00080

VI. VISUAL SERVOING

Our visual servoing strategy is a position-based approach that iteratively minimizes the offset between the detected gap center and the camera's optical center. This process utilizes a Proportional (P) control mechanism to generate translational movements that align the drone with the target in the lateral (Y) and vertical (Z) world axes.

A. Error Calculation and Coordinate Mapping

The first step is to quantify the error between the detected gap centroid (c_x, c_y) and the camera's image center $(\text{img\_c}_x, \text{img\_c}_y)$. The differences are calculated in pixel space as d_x (horizontal error) and d_y (vertical error).

A critical step is the mapping of the image-space error to the world coordinate system, where the drone's lateral motion is controlled by the Y-axis and its vertical motion by the Z-axis.

- The horizontal pixel error (d_x) is directly mapped to the required **lateral (Y-axis)** movement (d_{y_world}).
- The vertical pixel error (d_y) is directly mapped to the required **vertical (Z-axis)** movement (d_{z_world}).

B. Proportional Control Command Generation

To translate the pixel error into a stable and proportional control command, we apply tuned scaling factors (lateral_scale_y and lateral_scale_z). These constants define the gain of the proportional controller, determining the magnitude of the world adjustment relative to the observed pixel error. A negative sign is introduced to ensure that a positive pixel error (target to the right or below) results in a corrective movement in the negative world direction (left or down).

$$\text{d}_{y\_\text{world}} = -\text{d}_x \times \text{lateral\_scale}_y$$
$$\text{d}_{z\_\text{world}} = -\text{d}_y \times \text{lateral\_scale}_z$$

C. Target Waypoint Update and Execution

The calculated world offsets are used to define a new intermediate target position (Center_targetPose) relative to the drone's current world position (currentPose). Specifically, the d_{y_world} is used to update the Y-coordinate (lateral alignment). This new target waypoint is then executed using a fixed velocity of $0.31\,\text{m/s}$. This iterative movement ensures that the drone continuously shifts its world position until the gap centroid is centered in the image frame, at which point the errors (d_x and d_y) approach zero.

VII. ASSUMPTIONS

- It is assumed that Quadrotor is a point object.
- We have decoupled the camera orientation from the drone's body so that the Splat view always stays upright by sending only the yaw (heading) angle to the renderer while keeping roll and pitch fixed at zero. This prevents the video from flipping when the drone starts inverted, without affecting its dynamics or control behavior.

VIII. RESULTS

- Real time taken for simulation and rendering - 5:36 seconds
- 1Sim time for complete obstacle course - 1:38 seconds

IX. CONCLUSION

We were able to pass through the known and unknown gap with the drone. We were able to successfully return to start position after crossing the unknown gap.

X. Supplementary Material

The videos of the Assignment 5 are given along with both detections overlay and without overlay: https://bit.ly/44oBtM6

References

[1] https://github.com/milesial/Pytorch-UNet

[2] Ramana's Blender Tutorial

[3] Farnebäck, G. (2003). Two-Frame Motion Estimation Based on Polynomial Expansion. In: Bigun, J., Gustavsson, T. (eds) Image Analysis. SCIA 2003. Lecture Notes in Computer Science, vol 2749. Springer, Berlin, Heidelberg.

[4] Kroeger, T., Timofte, R., Dai, D. and Van Gool, L., 2016, September. Fast optical flow using dense inverse search. In European conference on computer vision (pp. 471-488). Cham: Springer International Publishing.

[5] Teed, Zachary, and Jia Deng. "Raft: Recurrent all-pairs field transforms for optical flow." Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16. Springer International Publishing, 2020.

[6] T. Lee, M. Leok and N. H. McClamroch, "Geometric tracking control of a quadrotor UAV on SE(3)," 49th IEEE Conference on Decision and Control (CDC), Atlanta, GA, USA, 2010, pp. 5420-5425, doi: 10.1109/CDC.2010.5717652.