RBE595 - Project 3 - Constrained Perception [using 4 Late Days]

Pranay Katyal
Worcester Polytechnic Institute
Worcester, MA, USA
pkatyal@wpi.edu

Anirudh Ramanathan
Worcester Polytechnic Institute
Worcester, MA, USA
aramanathan@wpi.edu

Hudson Kortus
Worcester Polytechnic Institute
Worcester, MA, USA
hkortus@wpi.edu

***Abstract*—The goal of this project was to develop a perception stack capable of detecting window frames within the VizFlyt simulation environment and enabling autonomous navigation through them using an existing navigation stack. A neural network was employed to train the perception system to perform semantic segmentation, allowing it to isolate and identify window frames from the surrounding environment. An error-correction control loop was integrated to continuously align the drone's sensor plane with the center of the nearest detected window, ensuring accurate orientation and successful traversal.**

I. Problem Statement

In this project, we extend our previous work on the navigation stack by developing and integrating a perception stack that enables the simulated quadrotor to perceive and interpret its environment before executing navigation tasks. While the navigation stack was responsible for path planning, trajectory generation, and control, the perception stack provides the necessary situational awareness that allows the system to identify goals and inform motion planning decisions.

The objective of this assignment is to implement a perception-driven navigation system in the VizFlyt photorealistic quadrotor simulator, where the drone must autonomously detect and fly through a sequence of checkpoints represented as rectangular windows. In each simulation scenario, up to three windows are randomly positioned and oriented within the environment. Although the specific dimensions of the windows are not provided, they remain constant across all instances. Each window can be visually recognized by its red frame, white checker pattern at the corners, and the WPI and Pear logos symmetrically placed along its borders.

The VizFlyt-simulated quadrotor is equipped with an RGB-D depth camera capable of generating dense depth maps of the scene ahead. However, the use of depth data was optional, and perception methods that operated solely on RGB visual input were encouraged for their generality and robustness to unseen environments. The overarching goal is for the drone to autonomously identify the nearest visible window in its field of view, determine its spatial relationship, and navigate through it by leveraging the previously developed navigation stack for trajectory tracking and collision avoidance within the VizFlyt simulation environment.

Fig. 1. Tello Drone

II. Methodology

The assignment simulated the motion of an aerial robot with the dynamics of a DJI Tello drone [1]. The drone dimensions were found from [2] and state the the drone's length, width, and height are 98, 95, and 41mm respectively. For the ease of simulation we simplified to drone to be a 100x100x50mm cuboid. We chose the slightly inflated x,y dimensions primary for convenience and extra safety margin. We chose to significantly increase the z height of the quadcopter because if an object came this close to the top or bottom of it in real life it would alter the propeller aerodynamics enough to cause a crash.

A. Window Detection

The RGB-D sensor onboard the simulated drone provides a continuous stream of image frames as the vehicle navigates through the environment. The primary objective within these images was the detection of the window frames that served as traversal checkpoints. To enable robust detection, environmental noise and irrelevant features needed to be abstracted from the image data. Traditional thresholding or color-based feature extraction techniques could be applied to isolate the window frames; however, such approaches require prior knowledge of

specific color or geometric characteristics, which are subject to significant variation due to changing lighting conditions and the drone's perspective during flight.

To generalize detection across varying environmental and visual conditions, a neural network-based perception approach was implemented. The network was trained to perform semantic segmentation by taking the raw RGB-D image from the sensor as input and producing a binary image that highlighted the window frame regions.

B. Blender Scene Rendering

To train this model, a large synthetic dataset was generated using a custom Blender script that produced randomized scenes containing windows in varying positions and orientations. Each sample in the dataset included both the rendered RGB-D image and its corresponding ground-truth binary segmentation mask. To enhance generalization and improve robustness, data augmentation techniques were applied, including color variation, background noise addition, and scaling of window proximity. These augmentations ensured that the trained model could adapt to a wide range of lighting and spatial conditions encountered in simulation.

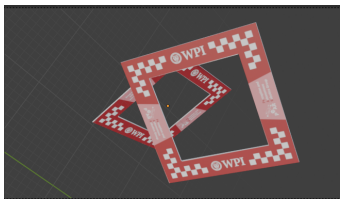

Fig. 2. Window Frame Scene in Blender

Fig. 3. Window Frame Scene in Blender (After background Removing)

C. Window Segmentation Model

To achieve robust semantic segmentation of windows, the group chose a classic U-Net [3]. A U-Net was selected due to its ubiquitous use in segmentation tasks and its data-efficient training. The U-Net features a downward contracting section where images are shrunk using max pooling and channels are doubled during convolution, a bottleneck section with a small image size and numerous channels, and an upward section where images are reconstructed through deconvolution (transposed convolution). Layers in the upward section are concatenated with their downward equivalents to restore context and accelerate training 4.

The model was trained for 25 epochs using Binary Cross-Entropy loss and optimized with Adam, with a weight decay of 0.001 and a learning rate of 4e-6.

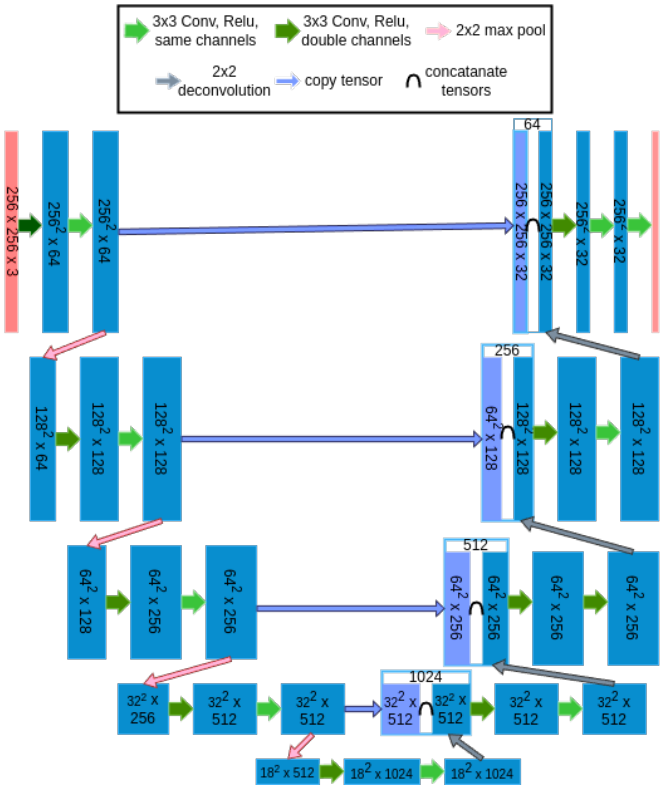

Fig. 4. Unet Architecture

Training data from Blender was augmented in the dataloader to help bridge the sim-to-real gap. Training data was stored as transparent PNGs for efficient creation and storage. When the model loaded an image, a random background was added, followed by a random set of the following augmentations: Gaussian noise was added across all color channels to make the model robust against common camera noise from underexposed images and cheap sensors; blur was added to simulate out-of-focus angles and low-quality optics; color jitter was introduced to simulate different camera sensors in different conditions; images were elastically transformed to simulate the wavy effects seen in Gaussian splats; and finally, images were center-cropped by a random amount to simulate varying zoom levels.

Finding the correct model hyperparameters was easy, but determining the ranges over which to randomly augment images in order to achieve good results in VizFlyt was a major challenge. Final model training results are shown in Figure 6,7,8,9 where the left image is the augmented model input, center image is ground truth, and right image is model prediction. Augmentation was very extreme, and in many of the cases its very difficult to for us to find the frame, but the model did a great job.

D. Perception Stack

Once the U-Net model achieved reliable segmentation accuracy, it was integrated into a perception controller designed to process window predictions in real time. The largest blob

Fig. 5. Graph of validation loss over each epoch showing no significant signs of overfitting

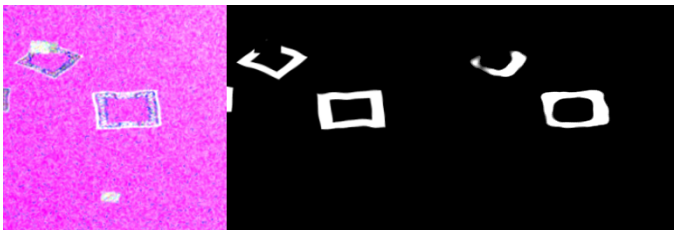

Fig. 6. Model training preview showing augmented input image on left, ground truth in middle, and predicted result on validation image on right

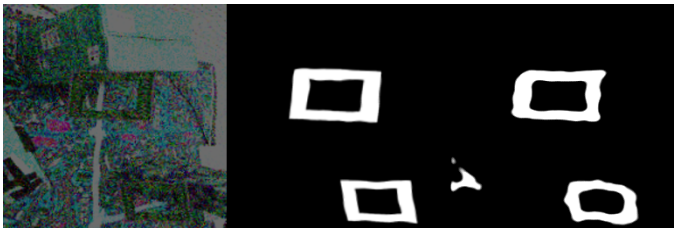

Fig. 7. Model training preview showing augmented input image on left, ground truth in middle, and predicted result on validation image on right

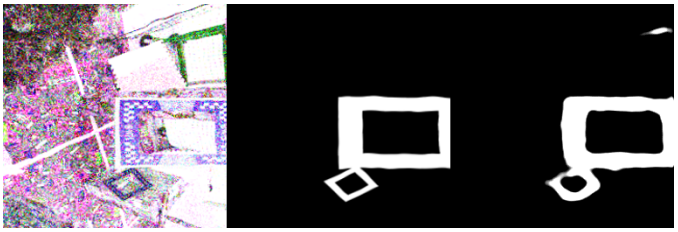

Fig. 8. Model training preview showing augmented input image on left, ground truth in middle, and predicted result on validation image on right

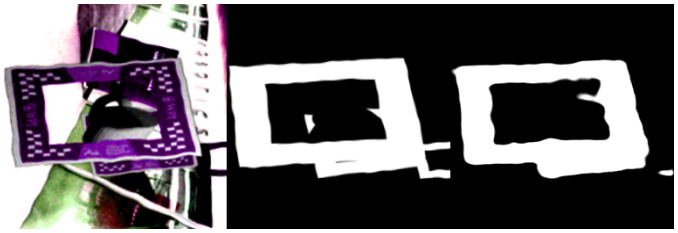

Fig. 9. Model training preview showing augmented input image on bottom, ground truth in middle, and predicted result on validation image on top

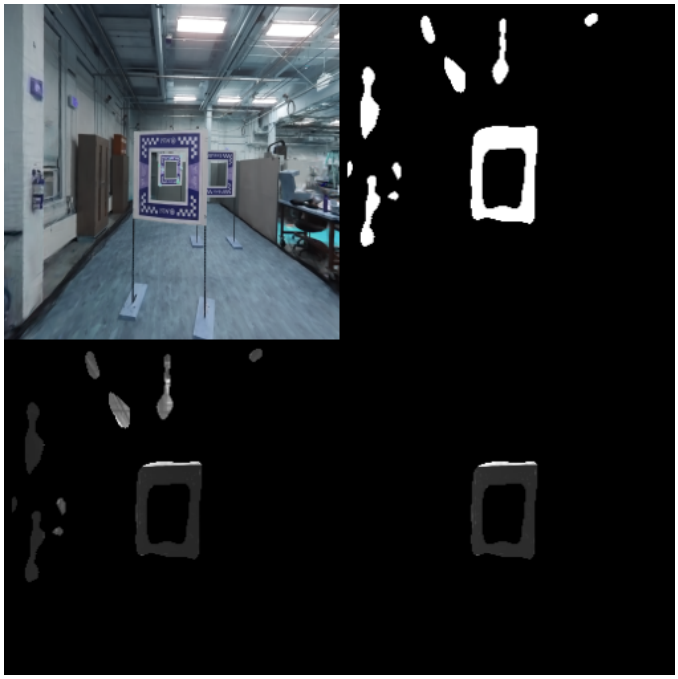

Fig. 10. Example of image processing pipeline: RGB input (top left), model prediction (top right), masked depth values (bottom left), and finally depth image of just the frame (bottom right).

in the model output was selected as the closest frame because all windows are the same physical dimension, so the apparent size of a window frame in the image correlated with its proximity to the sensor plane. The OpenCV command `cv2.connectedComponentsWithStats()` provided not only a list of blobs in the image sorted by size but also their centroids, which were crucial for estimating the drone's positional error relative to the window. This largest blob was masked with the metric depth image to get the drone's depth from the window. This process can be seen in Figure 10, where the RGB image is segmented using a U-Net, that output is used to mask the depth output, and then the largest blob is selected as the closest frame.

E. Visual Servoing

Once the pixel displacement of the center of the drone frame, as well as its depth, are found, we can calculate a control command to send to the controller using Equation (1).

$$\text{ctrl}_y = e_y, e_x, k_y \tag{1}$$

We multiply the pixel displacement e_y with the average metric depth of the frame e_x and multiply this by a scaling factor, which we experimentally derived to be 0.003. This scale is a crude analog for the camera intrinsic matrix.

These control inputs were appended to the world position of the drone and sent to the controller provided in the project. Once the camera was aligned with the center of the drone frame, the drone was commanded to approach to 200 cm away from the frame and realign itself before flying through.

This final alignment step was important to ensure the drone accounted for angled frames.

III. VizFlyt and Turing

This project utilized the VizFlyt simulator within the high-performance computing (HPC) environment provided by WPI's Turing cluster. The integration of VizFlyt with Turing enabled efficient execution of computationally intensive simulations with GPU acceleration.

A. Turing HPC Setup

We submitted jobs to the Turing cluster using a Slurm batch script `turing.sh`, which requests necessary resources and sets up the environment. The key script content is shown below:

We configured the request to use one node with 16 CPU cores, 32 GB RAM, and one GPU compatible with NVIDIA A30, H100, A100, or V100 models. The job wall-time limit was set to 1 hour.

To monitor the job status, the command `squeue --me` was used, allowing real-time tracking of personal running jobs on the cluster.

B. GPU Hardware

At runtime, the GPU resources available on Turing were as follows (example snapshot): The GPU used during simulation on the Turing cluster was an NVIDIA A30, as confirmed by the NVIDIA System Management Interface (nvidia-smi) output. The GPU driver version was 565.57.01, with CUDA version 12.7 installed. The GPU had 32000 MiB (24 GB) of memory available, was operating at a power level of 29 W out of a 165 W cap, and maintained a temperature of 33°C. At the time of the snapshot, no processes were running on the GPU, and it showed minimal memory usage and 0% utilization, indicating the system was idle or ready for workload execution.

This detailed information confirmed the use of an NVIDIA A30 GPU with 32 GB memory and CUDA version 12.7, providing ample computational power for GPU-accelerated VizFlyt rendering.

C. VizFlyt Simulator

VizFlyt is an open-source, perception-centric hardware-in-the-loop simulator tailored for autonomous aerial robot development [4]. It provides photorealistic first-person views and realistic flight dynamics within digitally mapped environments.

The simulator processes stitched 2D imagery and integrated map data to emulate complex robot-environment interactions. Visual outputs such as depth maps, RGBD frames, and matplotlib plots are saved as image sequences in distinct subfolders within a `Render` folder. These are later used for synchronized video creation, capturing both first-person and plan-view perspectives.

Our project tightly coupled the trajectory, planning ,and control stack with VizFlyt rendering pipeline running on the Turing cluster, enabling efficient simulation with real-time feedback and high-quality visualization.

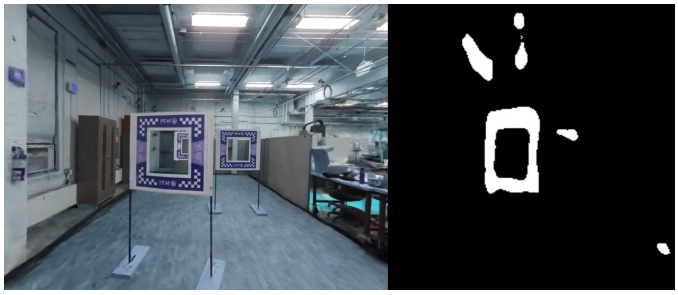

Fig. 11. Example of model prediction in VizFlyt

Fig. 12. Example of model prediction in VizFlyt

IV. Results

A. Model Results

11, 12 show the results of model predictions in VizFlyt. These predictions are noticeably worse than training. This is likely due to the lack of background images. Inexplicably, the model will only segment out the closest frame; this is not supposed to happen, and the model was not trained to do this. The model runs very fast with an input size of 256x256px, but its accuracy can be increased at the expense of run time if the input size is increased to 640x480px.

B. Perception Results

Even with the earlier outputs of the model training, the closest window was always the largest connected component in the frame, so it was correctly selected from the model output. This UNet was trained on 10,000 images produced in Blender with varying augmentations for even more data

This was further improved in the 25th epoch model (the one we are currently using), as compared to the 11th or 19th epoch, and it showcased significant improvements in segmenting out the windows. The Window Detector script, along with the Navigation script, then successfully used this largest connected component to align the Drone (Image frame) with its center. We then maintained this alignment and started moving towards it when it was small enough.

We also maintained a separate threshold that kept track of this alignment, making sure that as we moved towards it, we were then able to essentially enlarge the alignment that we couldn't infer previously. Once this threshold was crossed, we stopped the drone and ran the alignment again, making sure that it was keeping track of the middle.

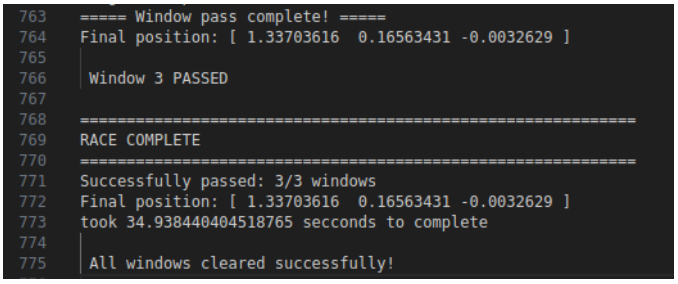

Fig. 13. Screenshot of the end of our simulation showing time took to compete

C. Race Time

After tuning the controller, velocities, and visual servoing heuristics, our drone was able to reliably complete the course in 35 seconds of sim time, 13.

V. Conclusion

To conclude, we were able to successfully make our simulated drone, run a perception model on it, which allowed it to infer what was in the scene, try finding a window in this scene, and then figure out with the detection script the center of the window.

Then the drone tries to align its center or here Image center, using visual servoing, with the PID controller, with the center of the window. Once the error is low enough, it starts moving towards it while making sure it's not deviating from the window center too much. Once it has crossed all 3 windows, the drone stops moving and the simulation ends.

Key accomplishments include:

- Goal reaching accuracy within 1 cm
- Successful implementation of UNet
- Successfully ran inferences in almost real time, on a Simulated environment.
- Successfully used Blender to create realistic ground truth data for Windows for training of UNet.
- An inference time of 21 ms was achieved.

A. Combined Video

The combined video for the P3 project can be found here.[5]. This video showcases a side-by-side view of the RGBD render and segmentation video.

References

[1] "Tello," https://store.dji.com/product/tello, DJI (Ryze Tech), 2025, accessed: 2025-09-30.

[2] "Tello specs," https://www.ryzerobotics.com/tello/specs, Ryze Tech, 2025, accessed: 2025-09-30.

[3] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," 2015. [Online]. Available: https://arxiv.org/abs/1505.04597

[4] K. Srivastava*, R. Kulkarni*, M. Velmurugan*, and N. J. Sanket, "Vizflyt: An open-source perception-centric hardware-in-the-loop framework for aerial robotics," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2025, accepted for publication. [Online]. Available: https://github.com/pearwpi/VizFlyt

[5] P. Katyal, A. Ramanathan, and H. Kortus, "Project files for rbe595 - project 3 - video," Google Drive, 2025, accessed: 2025-11-09. [Online]. Available: https://drive.google.com/drive/folders/1WFAFMGx0Xt5AfFqu6d4ih8-gEFDL7Bnm?usp=sharing