# PURSUIT EVASION STRATEGIES
for Drones in AirSim and Unreal Engine

Siddharth Anand | 20D070076

Guide: Prof. Debraj Chakraborty

Adaptive Camera Guided Control (ACGC)

Aim: To perform a real-time chase of the evader drone using the camera feed of the pursuer drone. The pursuer must estimate the direction of the evader and keep the evader visually trackable throughout the chase.

The objective is broken down into the following steps:

1. Compute the direction vector from the pursuer to the evader in the global frame using image-based estimation

2. Establish yaw control of the drone body to keep the evader horizontally centered in the camera image

3. Establish pitch (tilt) control of the gimbal-mounted camera to keep the evader vertically centered in the camera image

Note: Points 2 and 3 ensure that the evader remains within the field of view (FoV) of the pursuer drone camera

Camera Specifications for AirSim

- We use the `front_center` camera on the default `SimpleFlight` drone in `AirSim`
- The camera outputs images with:

$$W = 256 \text{ pixels (Width)}, \quad H = 144 \text{ pixels (Height)}$$

$$\theta_y = 90^\circ \text{ (Vertical Field of View)}$$

- The aspect ratio is:

$$a = \frac{W}{H} = \frac{256}{144} = \frac{16}{9}$$

- The focal lengths along each axis are:

$$f_y = \frac{H}{2 \cdot \tan(\theta_y/2)} = 72, \qquad f_x = a \cdot f_y = 128$$

- The principal point (image center) is:

$$x_c = \frac{W}{2} = 128, \quad y_c = \frac{H}{2} = 72$$

front_center camera

SimpleFlight Drone

AirSim Object Detection API

The AirSim Object Detection API detects predefined objects in the Unreal Engine environment using their unique object IDs

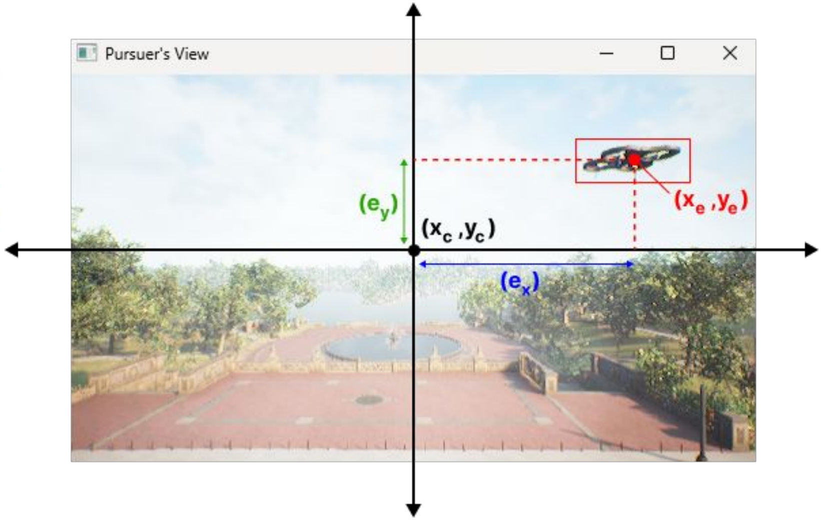
- Each object in the Unreal scene (e.g., the Evader drone) is assigned a unique `object ID`
- The `simGetDetections()` API retrieves the bounding box for each detected object, if visible within the camera's field of view
- Bounding Box Center Coordinates are defined as:

$$(x_e, y_e) = \left(\frac{x_{\min} + x_{\max}}{2}, \frac{y_{\min} + y_{\max}}{2} \right)$$

- Pixel Offset Errors are computed as:

$$e_x = x_e - x_c, \qquad e_y = y_e - y_c$$

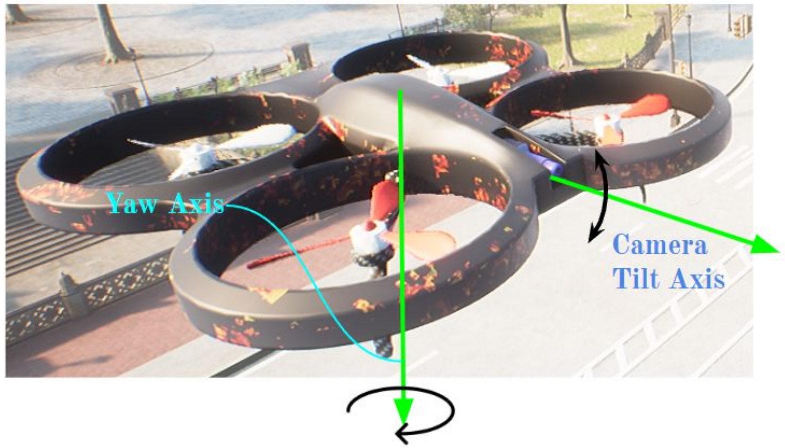
where x_c, y_c are the camera's principal point coordinates defined earlier



Yaw and Camera-Tilt Control using PID

- The drone's yaw (ψ_B) and the camera's tilt (θ_C) are separately controlled using two discrete-time PID controllers
- The objective is to keep the detected evader centered in the image by minimizing pixel offset errors
- Pixel offset errors are computed as:

$$e_x[t] = x_e[t] - x_c, \quad e_y[t] = y_e[t] - y_c$$



Drone body yaw axis and drone camera tilt axis

Yaw and Camera-Tilt Control using PID

- Yaw rate command (for drone body):

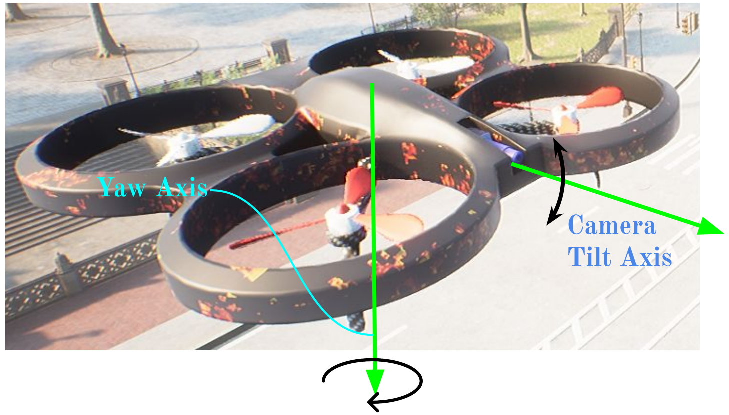
$$\dot{\psi}_B[t] = K_{p,\text{yaw}} \cdot e_x[t] + K_{i,\text{yaw}} \cdot \sum_{\tau=0}^{t} e_x[\tau] \cdot \Delta t + K_{d,\text{yaw}} \cdot \frac{e_x[t] - e_x[t-1]}{\Delta t}$$

- This yaw rate can be directly fed to the drone via a velocity command
- Pitch rate command (for camera):

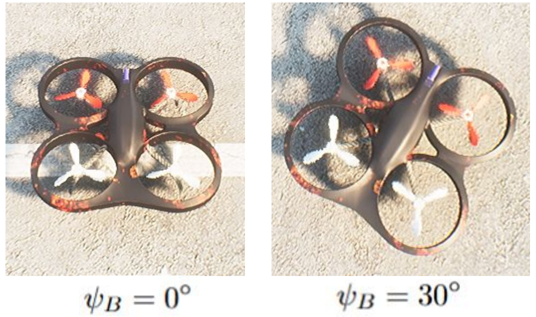
$$\dot{\theta}_C[t] = K_{p,\text{cam}} \cdot e_y[t] + K_{i,\text{cam}} \cdot \sum_{\tau=0}^{t} e_y[\tau] \cdot \Delta t + K_{d,\text{cam}} \cdot \frac{e_y[t] - e_y[t-1]}{\Delta t}$$

- Since AirSim does not allow direct rate commands for camera pitch, we integrate:

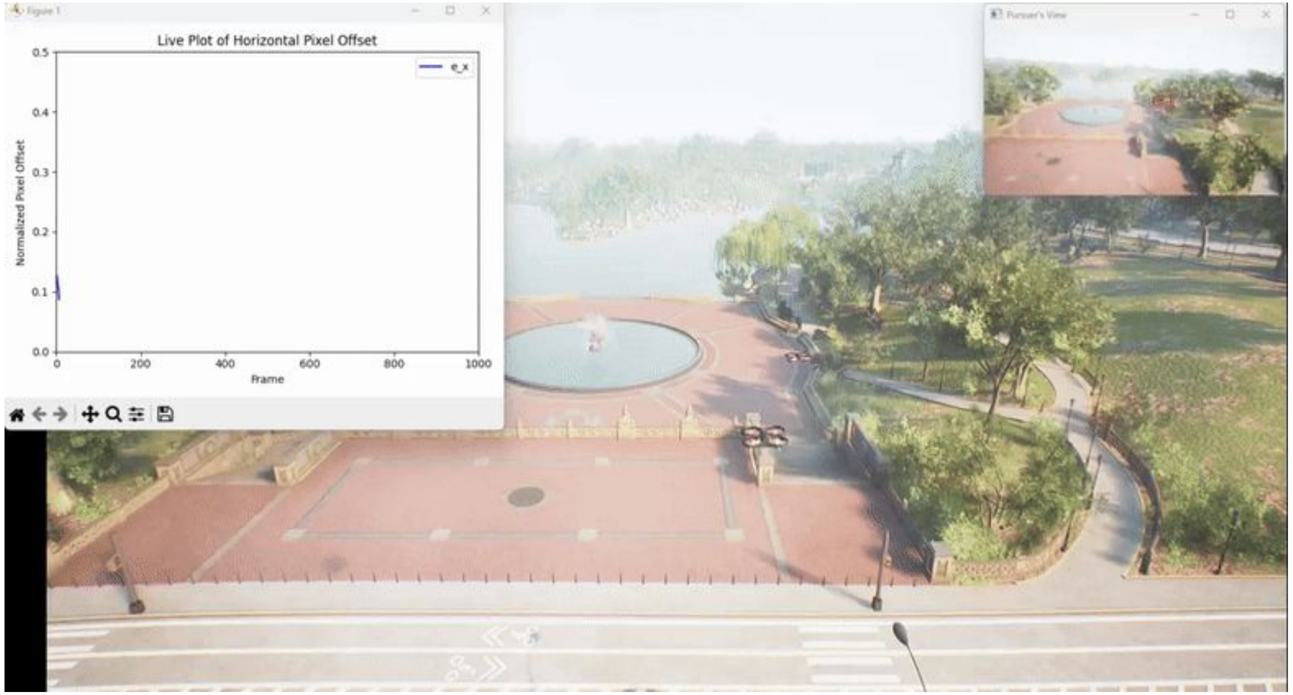
$$\theta_C[t] = \theta_C[t-1] + \dot{\theta}_C[t] \cdot \Delta t$$



Drone body yaw axis and drone camera tilt axis



Yaw Control Response to Horizontal Pixel Offset



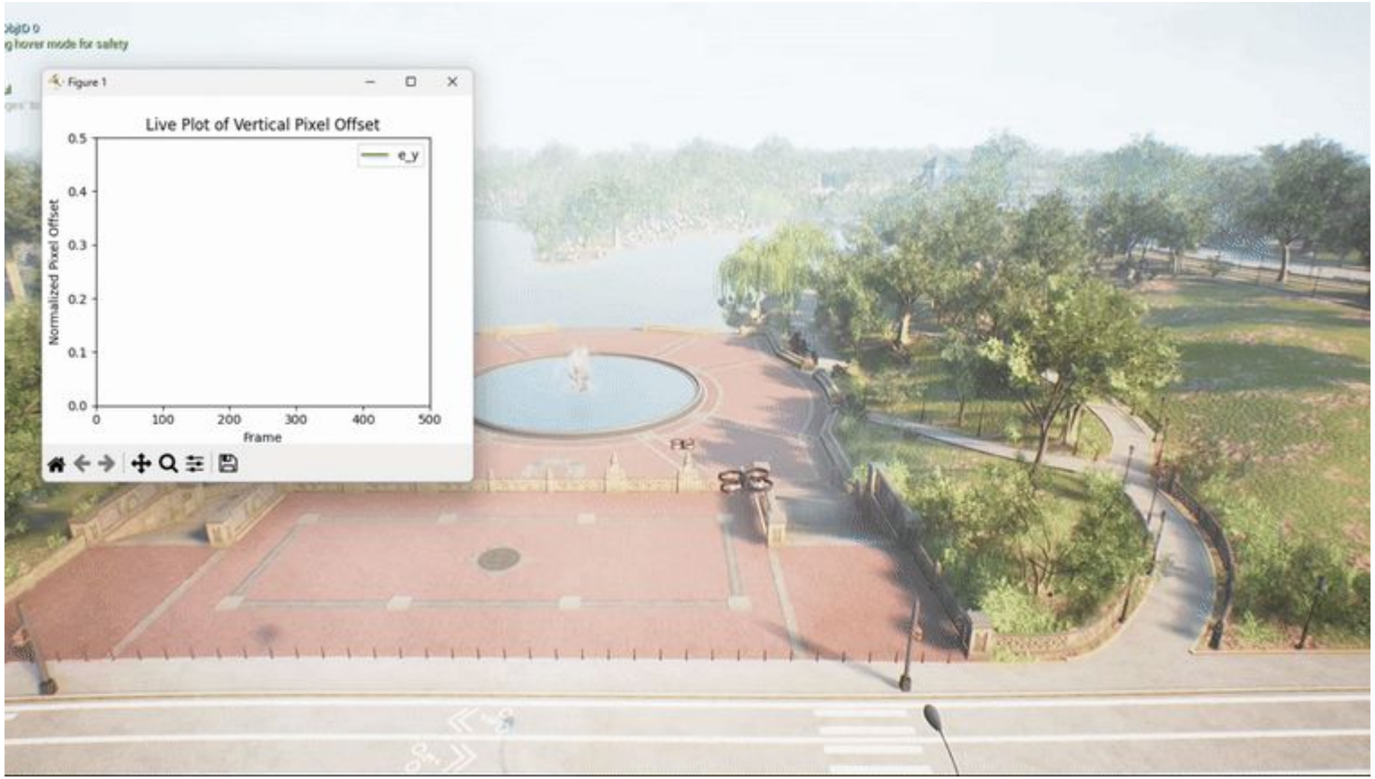
Gains used:

$$K_{p,\text{yaw}} = 0.5,$$

$$K_{i,\text{yaw}} = 0.02,$$

$$K_{d,\text{yaw}} = 0$$

Camera-Tilt Control Response to Vertical Pixel Offset



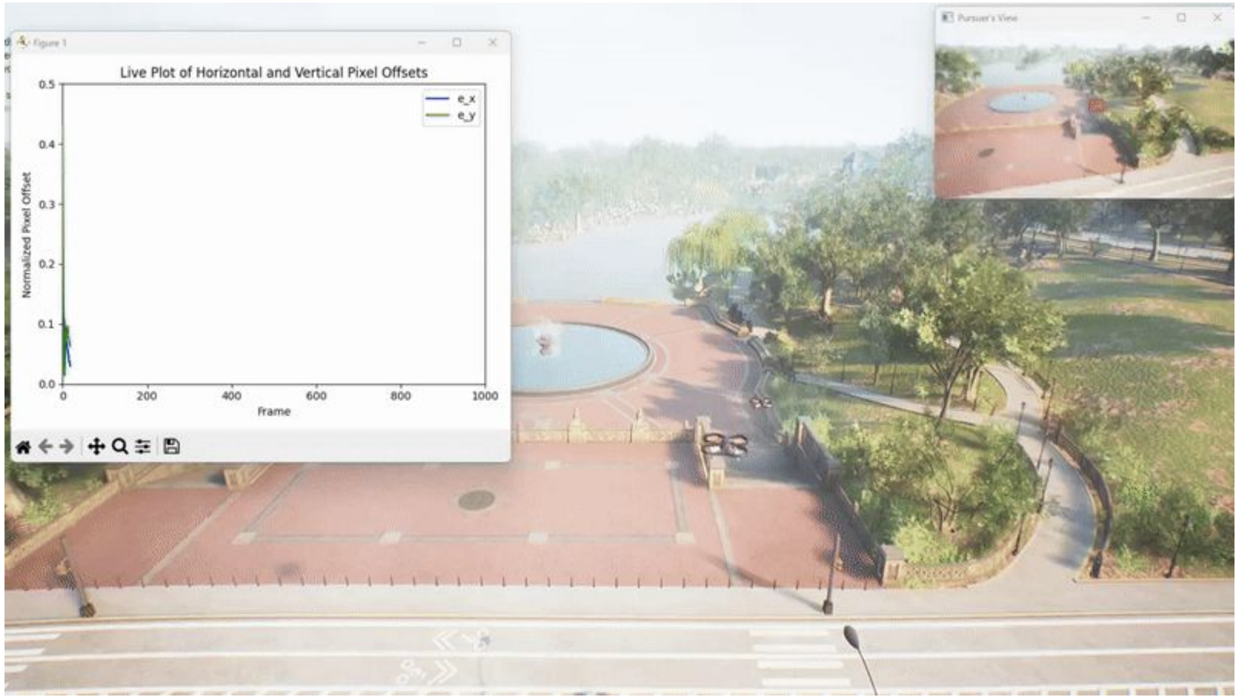
Gains used:

$$K_{p,\text{cam}} = 0.02,$$

$$K_{i,\text{cam}} = 0.002,$$

$$K_{d,\text{cam}} = 0$$

Full Visual Tracking using PID (Yaw and Camera Tilt Control)



Gains used:

$$K_{p,\text{yaw}} = 0.5,$$

$$K_{i,\text{yaw}} = 0.02,$$

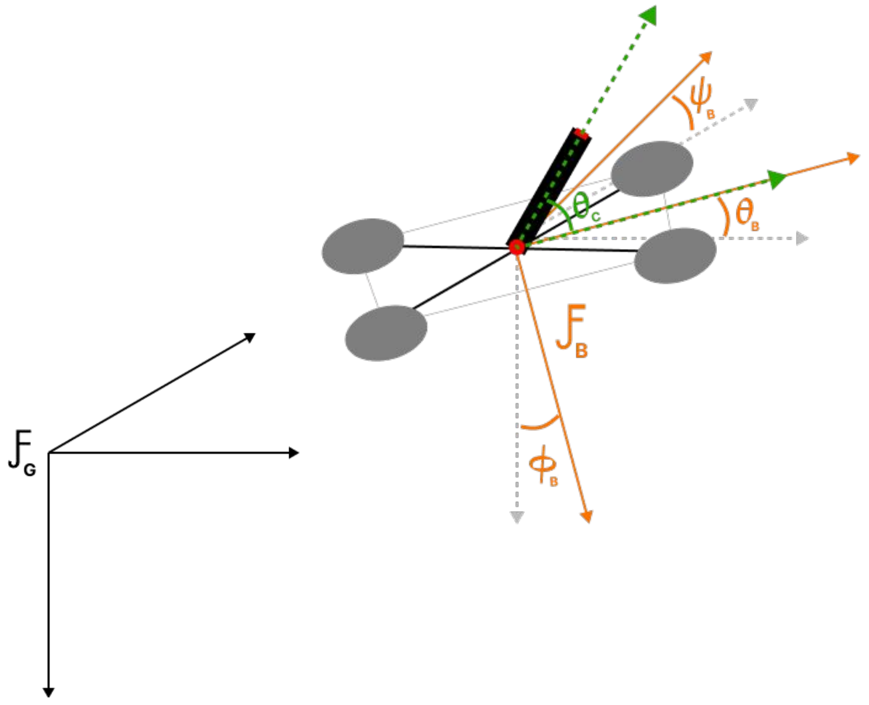
$$K_{d,\text{yaw}} = 0$$

$$K_{p,\text{cam}} = 0.02,$$

$$K_{i,\text{cam}} = 0.002,$$

$$K_{d,\text{cam}} = 0$$

Estimation of Heading Vector



Drone and camera orientation shown with body and global frames

Here we estimate the heading vector from the pursuer to the evader, expressed in the global coordinate frame

- The direction vector $\hat{\vec{d}}_c$ in the camera frame is first rotated into the **body frame** using:

$$\hat{\vec{d}}_b = R_{\text{cam}}^{\text{body}} \cdot \hat{\vec{d}}_c$$

where $R_{\text{cam}}^{\text{body}} = R_y(\theta_c)$, and θ_c is the tilt angle of the camera

- The rotation about the body-y axis is represented as:

$$R_y(\theta_c) = \begin{bmatrix} \cos\theta_c & 0 & \sin\theta_c \\ 0 & 1 & 0 \\ -\sin\theta_c & 0 & \cos\theta_c \end{bmatrix}$$

- The drone's orientation is given by a quaternion $q = (w, x, y, z)$, which converts to a rotation matrix:

$$R_{\text{body}}^{\text{global}} = R(q)$$

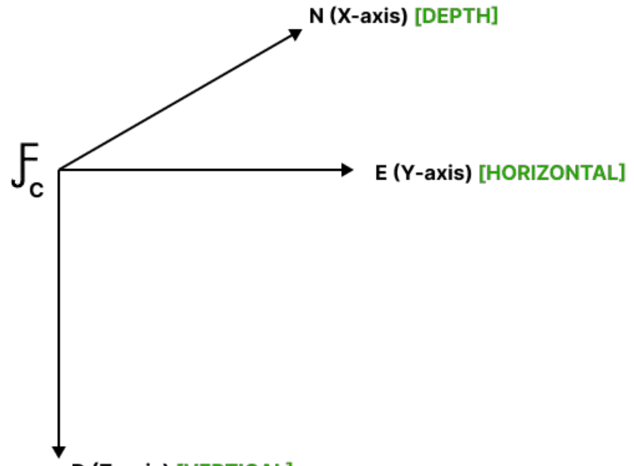
$$R(q) = \begin{bmatrix} 1-2(y^2+z^2) & 2(xy-zw) & 2(xz+yw) \\ 2(xy+zw) & 1-2(x^2+z^2) & 2(yz-xw) \\ 2(xz-yw) & 2(yz+xw) & 1-2(x^2+y^2) \end{bmatrix}$$

- The final heading vector in the **global frame** is computed as:

$$\hat{\vec{d}}_g = R_{\text{body}}^{\text{global}} \cdot \hat{\vec{d}}_b$$

and is used for giving velocity commands to the drone

Estimation of Relative Direction Vector in Camera Frame



Camera Frame (NED Coordinates)

- Let D be the depth at the pixel location (x_e, y_e) obtained from the depth image
- The direction vector to the evader drone (from the pursuer) in the **camera frame** is given by:

$$\hat{\vec{d}}_c = \begin{bmatrix} D & e_x \cdot \frac{D}{f_x} & -e_y \cdot \frac{D}{f_y} \end{bmatrix}^\top$$

- Using the standard pinhole camera model, with image center at (x_c, y_c), we have:

$$\begin{bmatrix} x_e \\ y_e \end{bmatrix} = \begin{bmatrix} \frac{f_x H}{D} + x_c \\ \frac{f_y(-V)}{D} + y_c \end{bmatrix} \Rightarrow \begin{bmatrix} e_x \\ e_y \end{bmatrix} = \begin{bmatrix} x_e - x_c \\ y_e - y_c \end{bmatrix}$$

- Substituting and rearranging gives:

$$H = \frac{De_x}{f_x}, \quad V = \frac{-De_y}{f_y}$$

- H and V represent the horizontal and vertical displacements in the camera frame respectively

But why do we need Depth?

- Isn't direction all we care about?
  Why not just use unit vectors?
- If we normalize the heading vector, doesn't depth D cancel out anyway?
- Can't we chase purely based on image-plane offsets (e_x, e_y)?

Adaptive Power Strategy for LOS Chase

- **Power** refers to the *magnitude* of the heading direction vector used to control the pursuer drone's body velocity
- It directly scales the pursuer's speed in the direction of the estimated heading vector
- In static power strategies, this magnitude is fixed — typically using unit vectors — and hence does not require depth information
- However, a fixed-speed chase may cause overshooting or sluggish pursuit depending on distance
- For more effective and flexible chasing, we define an **adaptive power** function that varies with the estimated depth D (or $\|\hat{\vec{d}}_g\|$)
- This allows the drone to chase more aggressively when farther from the evader and slow down as it gets closer, reducing overshoot and improving stability

Adaptive Power for LOS Chase

Logarithmic Power Scaling Function:

$$p = p_{\min} + \left(\frac{\log(1 + \|\hat{\vec{d}}_g\|)}{\log(1 + \|\hat{\vec{d}}_g\|) + 1} \right) \cdot (p_{\max} - p_{\min})$$

For our application:

- $p_{\min} = 1$ (Evader speed)
- $p_{\max} = 2.5$

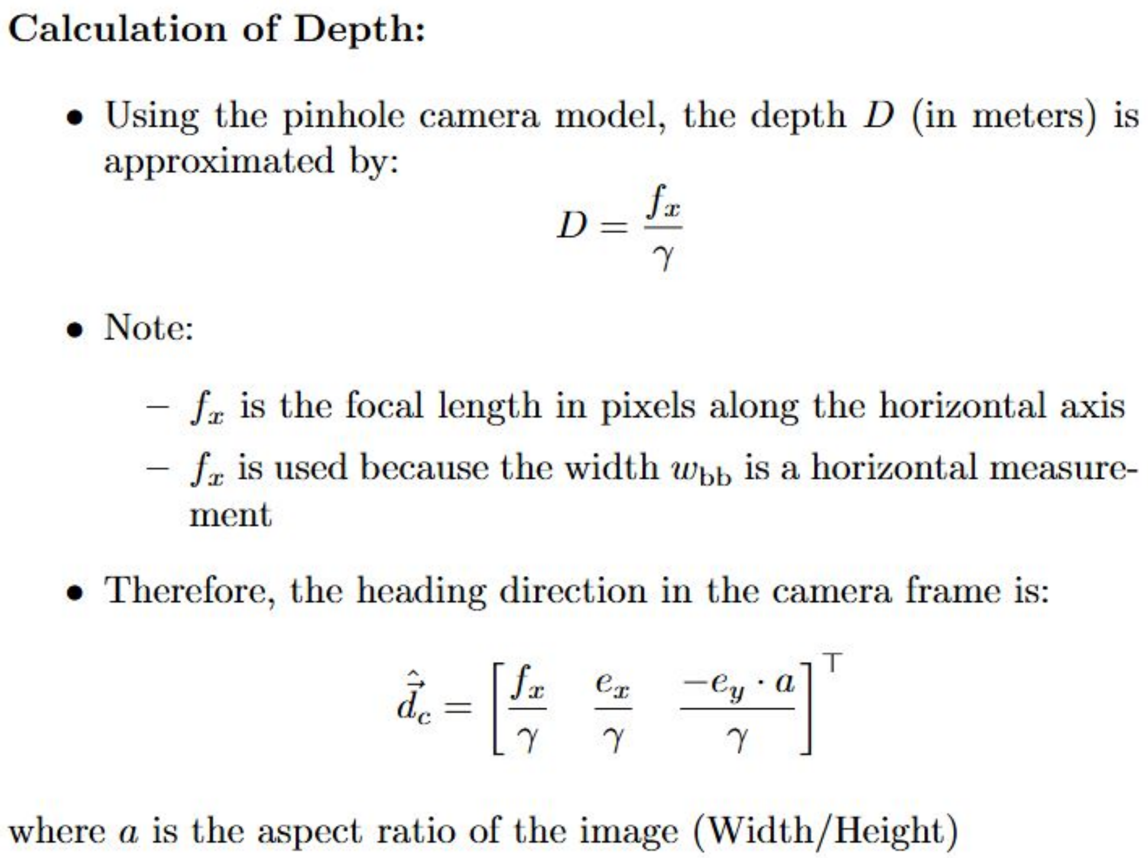
$$p = 1 + \left(\frac{\log(1 + \|\hat{\vec{d}}_g\|)}{\log(1 + \|\hat{\vec{d}}_g\|) + 1} \right) \cdot 1.5$$

Strategy 1: Depth Estimation using Scaling Factor Based Approach

- We define the bounding box width in the image as: $w_{\text{bb}} = x_{\text{max}} - x_{\text{min}}$ the difference of rightmost and leftmost pixel coordinates of the bounding box
- This width approximates the apparent pixel width of the evader drone in the image
- The actual width of the drone is approximately 1.36 m
- We define a **scaling factor** γ to convert pixel width to metric width:

$$\gamma = \frac{w_{\text{bb}}}{1.36}$$

Strategy 1: Depth Estimation using Scaling Factor Based Approach



Calculation of Depth:

- Using the pinhole camera model, the depth D (in meters) is approximated by:

$$D = \frac{f_x}{\gamma}$$

- Note:
  - f_x is the focal length in pixels along the horizontal axis
  - f_x is used because the width w_{bb} is a horizontal measurement
- Therefore, the heading direction in the camera frame is:

$$\hat{\vec{d}}_c = \begin{bmatrix} \frac{f_x}{\gamma} & \frac{e_x}{\gamma} & \frac{-e_y \cdot a}{\gamma} \end{bmatrix}^\top$$

where a is the aspect ratio of the image (Width/Height)

Tracking Metrics

- **Live Angle Error:** Angle between the estimated heading vector $\hat{\vec{d}}_g$ and the true heading vector $\vec{d}_g$ (in degrees)

$$\text{Angle Error} = \cos^{-1}\left(\frac{\hat{\vec{d}}_g \cdot \vec{d}_g}{\|\hat{\vec{d}}_g\| \cdot \|\vec{d}_g\|}\right)$$

- **True Distance to Evader:** Norm of the ground-truth vector $\vec{d}_g$, i.e., the Euclidean distance from the pursuer to the evader in global coordinates

$$\|\vec{d}_g\| = \sqrt{(x_e - x_p)^2 + (y_e - y_p)^2 + (z_e - z_p)^2}$$

Strategy 1: Real-Time Chase Performance (using Adaptive Power)

```
l - ObjID 9
successful
d
ed, entering hover mode for safety
[illegible]
```

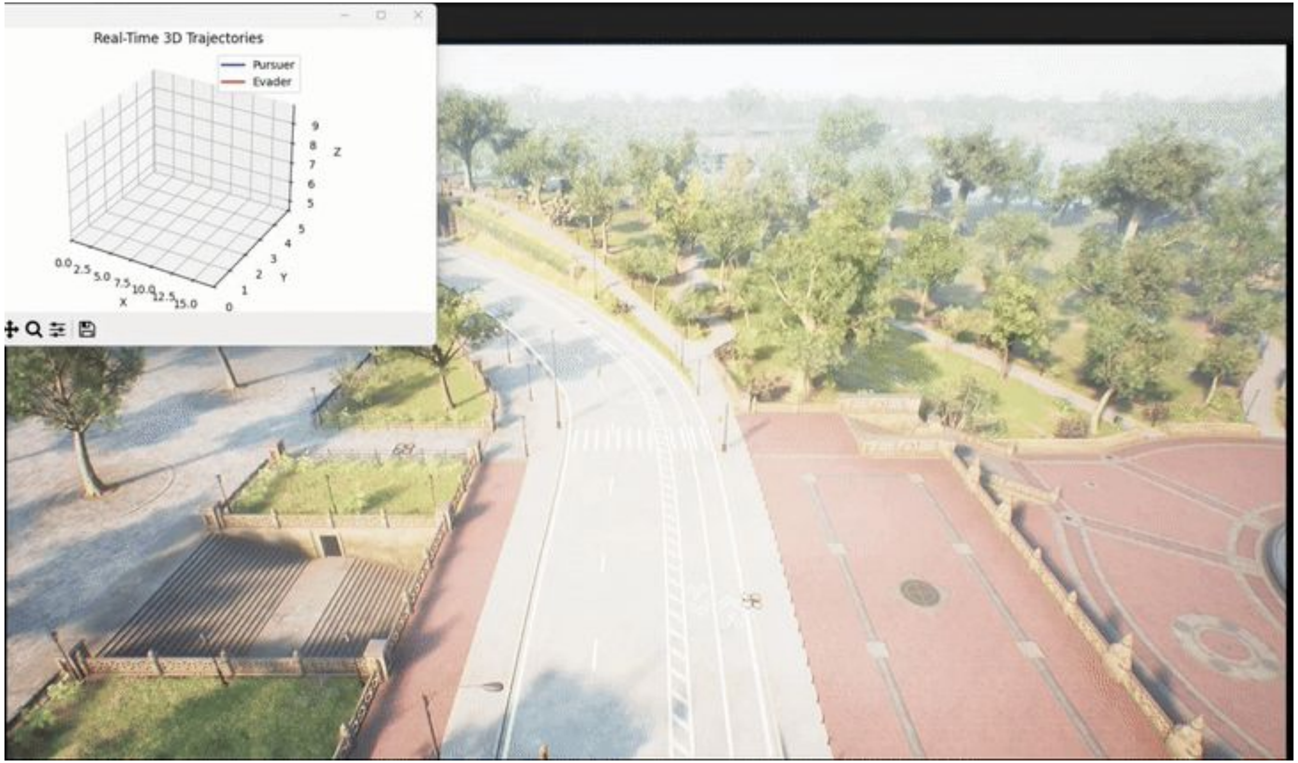
Strategy 2: Depth Estimation Using Depth Camera

- **AirSim's Depth Camera:** AirSim provides access to simulated depth data via a special camera mode called `DepthPerspective`, which mimics a pinhole camera capturing depth information
- **How it works:** Each pixel in the captured `DepthPerspective` image encodes the distance (depth) from the camera center to the first object it intersects along the corresponding viewing ray
- **Virtual Raycasting:** Internally, Unreal Engine performs raycasting per pixel: it projects rays from the camera center through each pixel and returns the distance to the first object hit. This forms a 2D depth map
- **Camera Setup:** The `DepthPerspective` feed is obtained from the same physical camera (e.g., `front_center`) as RGB or segmentation modes, but with a different imaging mode set. No separate physical camera is required

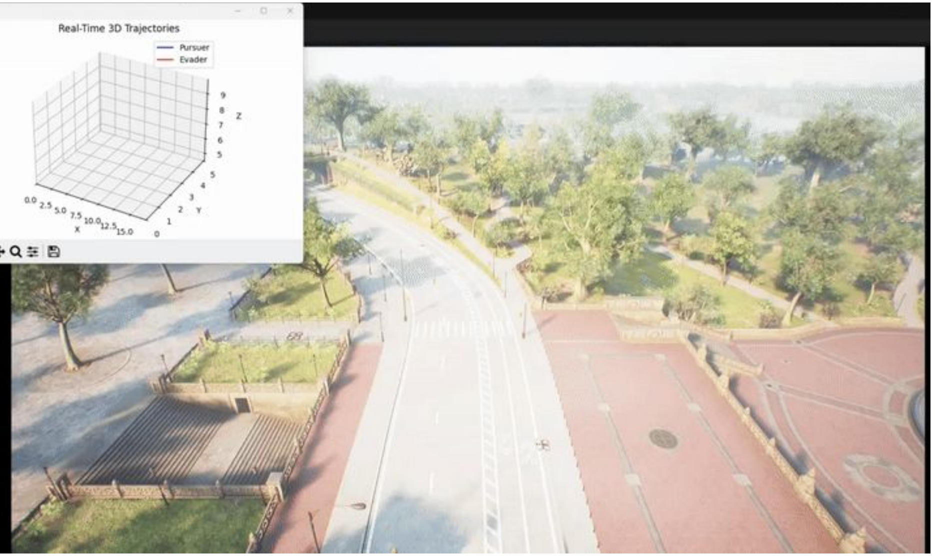
Strategy 2: Depth Estimation Using Depth Camera

- **Why it's useful:** This mode provides accurate depth estimates in meters at the pixel level, without needing explicit stereo computation or manual calibration
- **Limitation:** This approach only gives the depth to the *first visible surface* in the line of sight. Transparent or reflective surfaces can lead to incorrect readings
- **Conclusion:** The direction vector $\hat{\vec{d}}_c$ is estimated identically to Strategy 1, but now depth D is directly retrieved from AirSim's API rather than approximated using scaling

Strategy 2: Real-Time Chase Performance (using Adaptive Power)

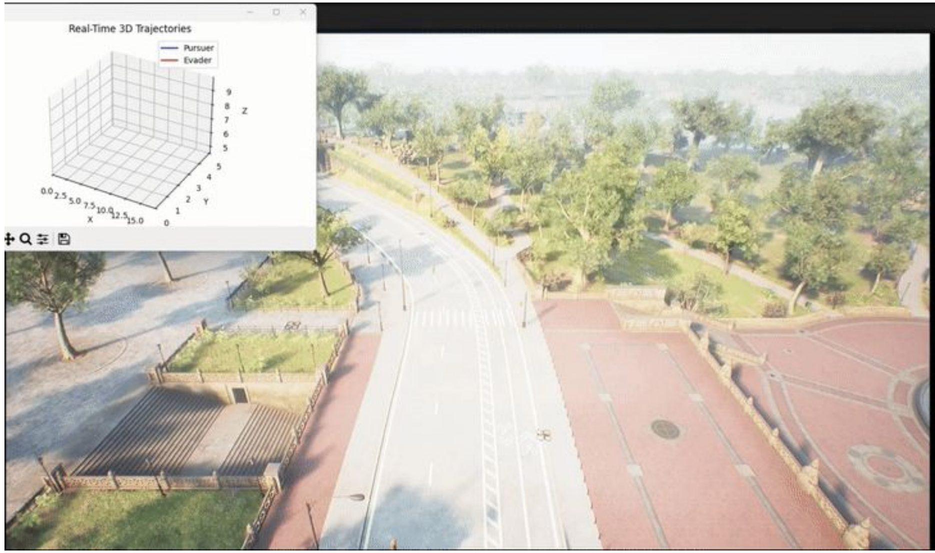


Adaptive Power vs Static Power Comparison



Adaptive Power
(chase time = 15.1 secs)

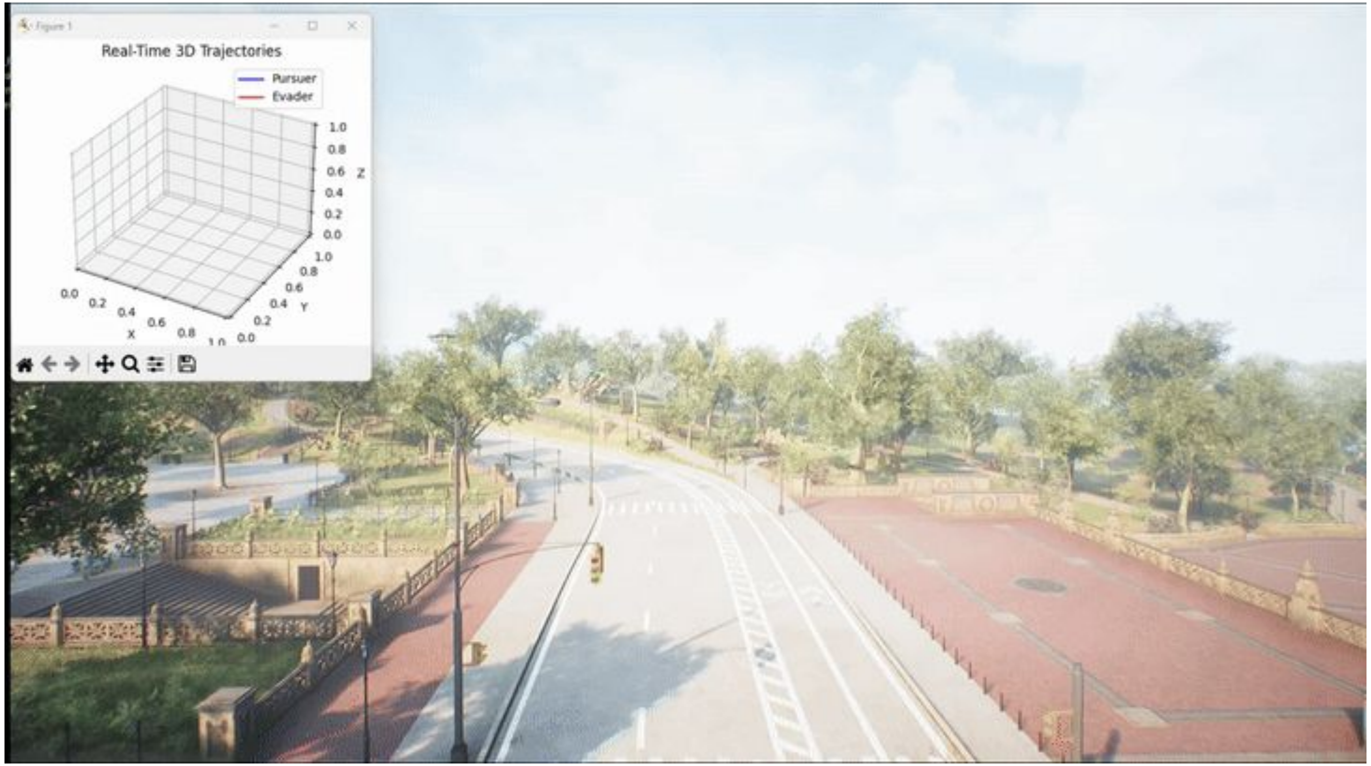
$$\boxed{p_{\text{min}} = 1 \text{ (Evader speed)} \quad p_{\text{max}} = 2.5}$$



Static Power
(chase time = 28.5 secs)

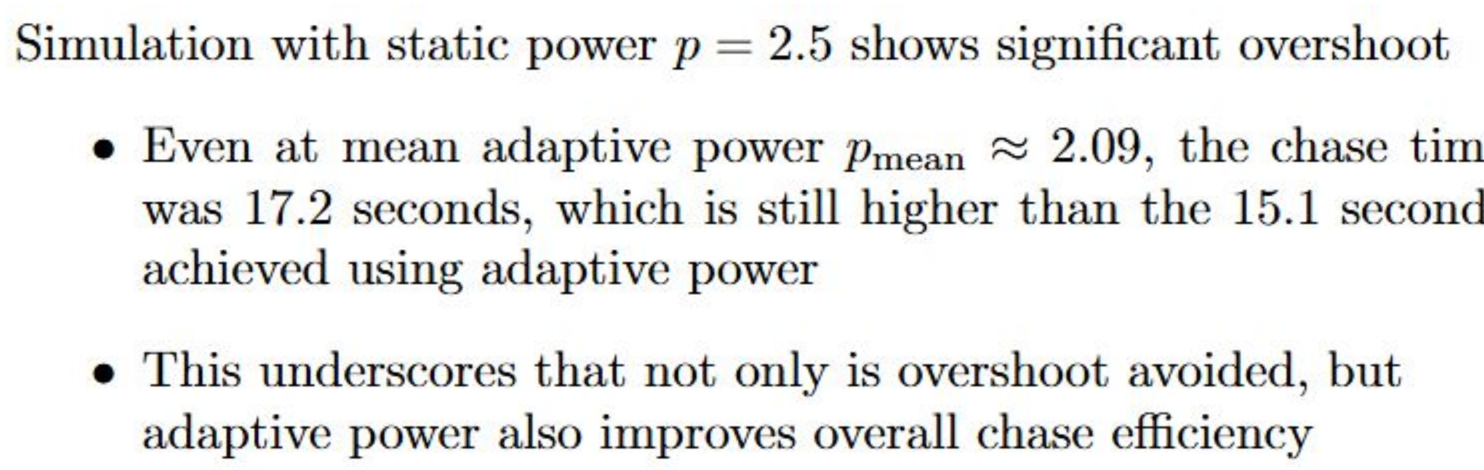
$$\boxed{p = 1.5}$$

Adaptive Power vs Static Power Comparison



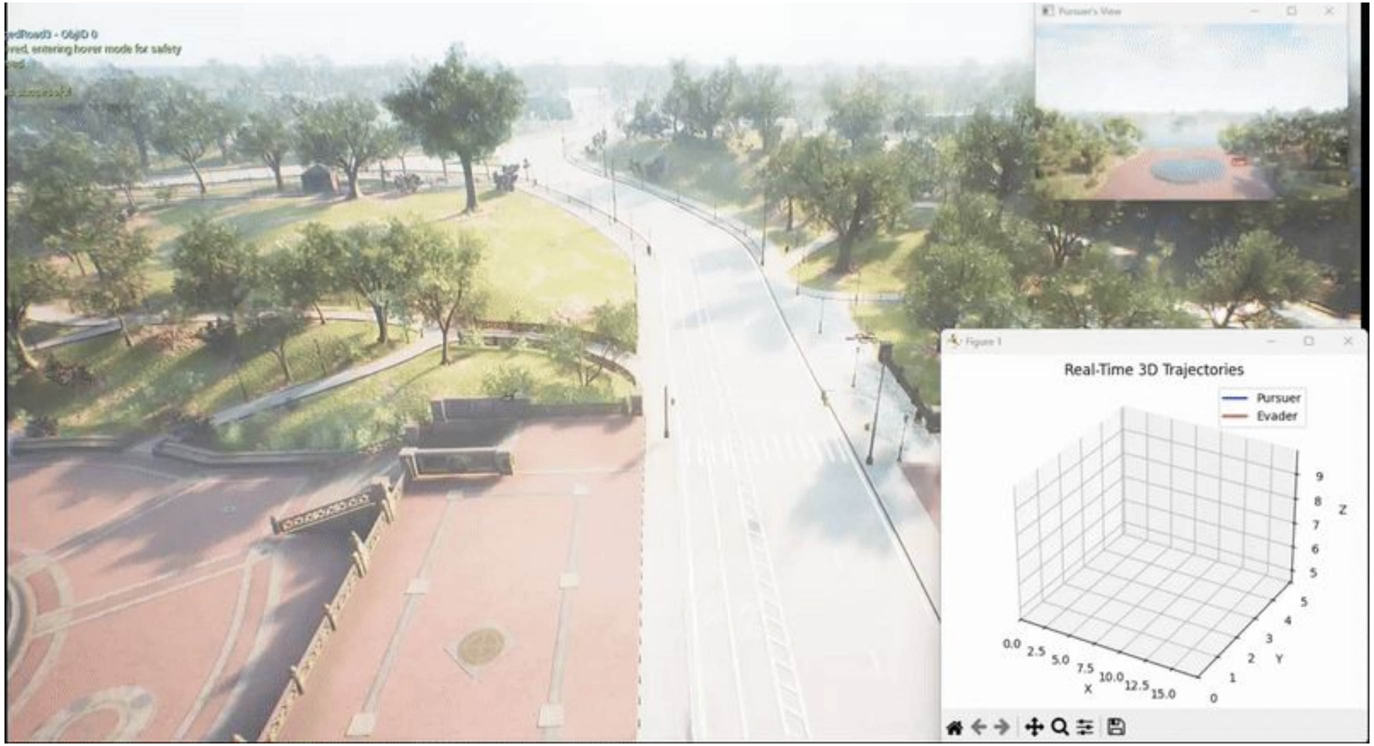
A simulation with static power $p = 2.5$ shows significant overshoot

Adaptive Power vs Static Power Comparison


Simulation with static power $p = 2.5$ shows significant overshoot

- Even at mean adaptive power $p_{\text{mean}} \approx 2.09$, the chase time was 17.2 seconds, which is still higher than the 15.1 seconds achieved using adaptive power
- This underscores that not only is overshoot avoided, but adaptive power also improves overall chase efficiency

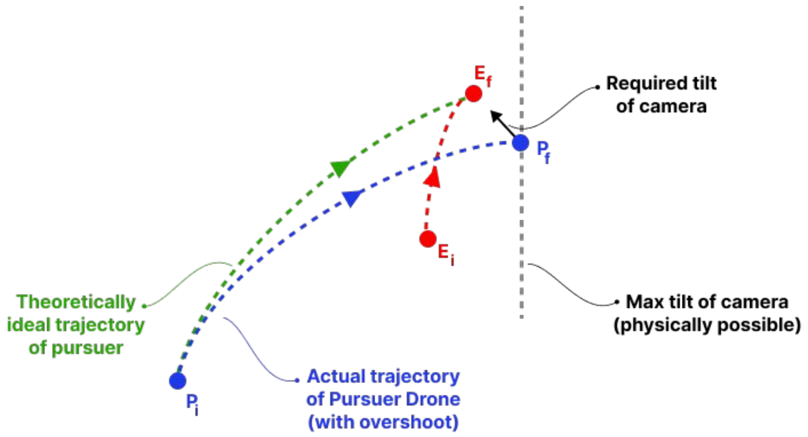
Gimbal locking due to Depth Overshoot



```
edRoad3 - ObjID 0
ved, entering hover mode for safety
ed
successful
```

Gimbal Locking Due to Depth Overshoot

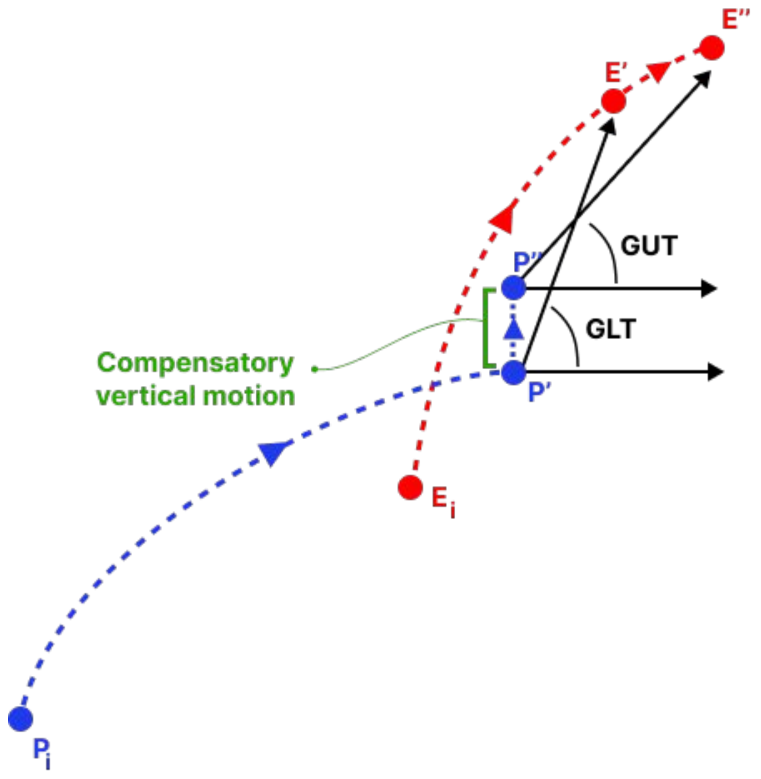
- Gimbal locking in the pursuer drone's camera arises when the evader moves vertically and the pursuer attempts to track it using excessive pitch (tilt) motion
- When the evader moves high in the camera's field of view (i.e., large e_y), the PID controller commands a strong upward tilt to re-center it
- If this correction is too aggressive — the required pitch can exceed the gimbal's physical limits, causing gimbal locking
- Specifically, when the evader is far above the pursuer, it can cause the **commanded** pitch angle $\theta_c > 90^\circ$, which is not physically realizable by the gimbal
- This leads to undefined or unstable camera behavior, referred to as **gimbal locking**



E_f
Required tilt of camera
P_f
E_i
Theoretically ideal trajectory of pursuer
Max tilt of camera (physically possible)
P_i
Actual trajectory of Pursuer Drone (with overshoot)

The diagram illustrates the ideal vs. overshooting motion of the pursuer under vertical evader movement.

Gimbal Locking Compensation



- At high camera tilt angles, the drone may approach a critical state where overshoot is likely
- This critical tilt is referred to as the **Gimbal Lock Threshold (GLT)**:

$$\theta_{\text{tilt}} \geq \text{GLT} \Rightarrow \text{Risk of gimbal locking}$$

- To prevent this, the drone is commanded to **increase altitude**:
  - The drone redirects its motion along the vertical axis (z direction)
  - This vertical climb continues until θ_{tilt} drops below a safer value
- This safer value is called the **Gimbal Unlock Threshold (GUT)**:

$$\theta_{\text{tilt}} \leq \text{GUT} \Rightarrow \text{Overshoot likely avoided}$$

- These thresholds are **experimentally determined** and depend on the chase aggressiveness (`Power`)

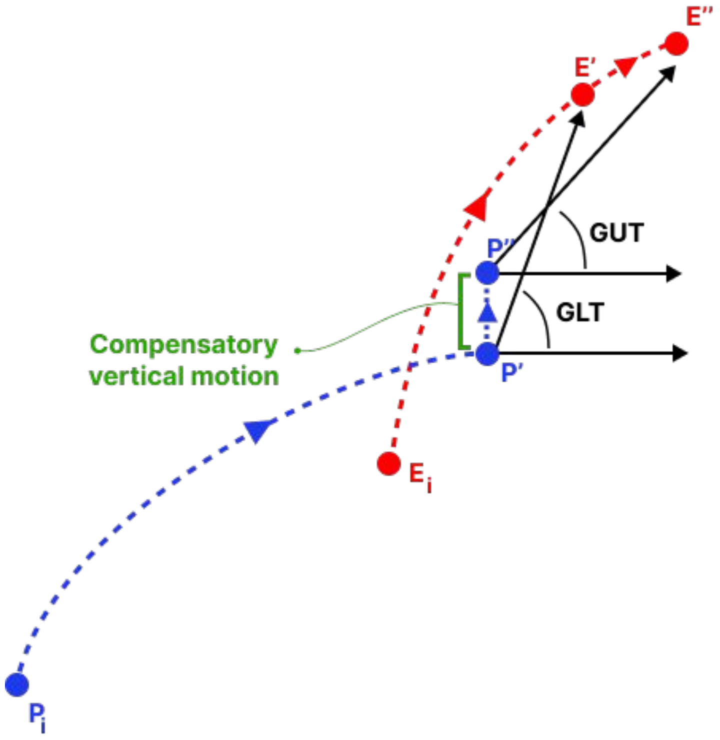
Gimbal Locking Compensation

- These thresholds are **experimentally determined** and depend on the chase aggressiveness (`Power`)

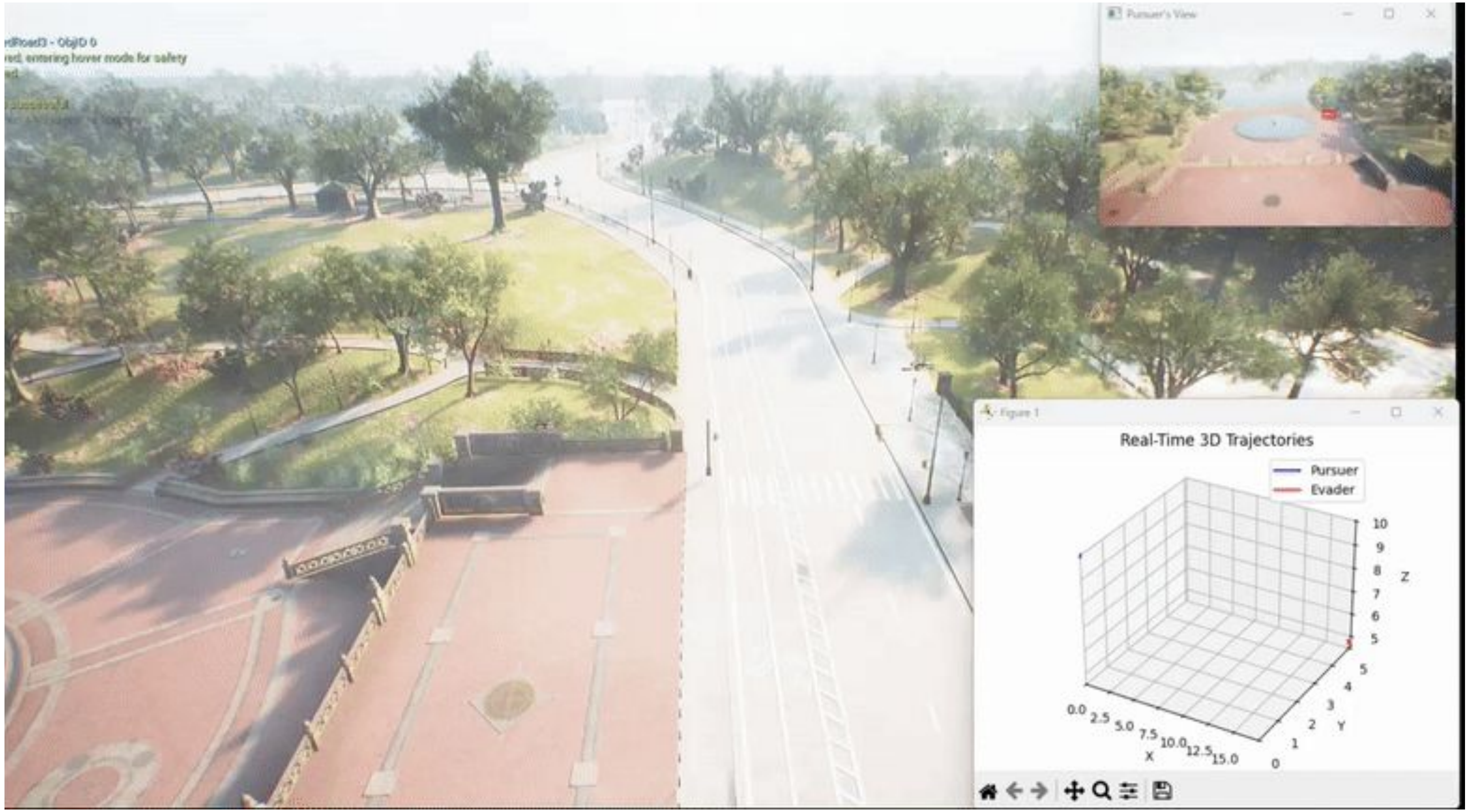
For `Power` $\in (1,\ 2.5)$, we have chosen following thresholds:

$$\text{GLT} = 70^\circ \quad , \qquad \text{GUT} = 60^\circ$$

Note: This discussion assumes the pursuer is *below* the evader. If the pursuer is *above* the evader, then a **decrease in altitude** is required to avoid gimbal locking.



Gimbal Locking Compensation



```
rdRoad3 - ObjID 0
red; entering hover mode for safety
ed
```

Voronoi Based Pursuit Strategy

- Theoretical formulation of this strategy was developed by **Deep Boliya** and **Adityaya Dhande**
- I implemented and tested the strategy in **AirSim**, validating its effectiveness through simulation

System Setup and Assumptions

- n pursuers $\{P_1, P_2, \ldots, P_n\}$ and one evader E operate in 3D space
- Each agent follows single integrator dynamics:

$$\dot{P}_i = u_i, \quad \dot{E} = u_E$$

- All agents are speed-constrained:

$$\|u_i\| \leq v_{max}, \quad \|u_E\| \leq v_{max}$$

- Objective: pursuers minimize capture time; evader maximizes time before capture
- Capture occurs if $\|E - P_i\| = 0$ for some i

Voronoi Region and Reachability

- The evader's **Voronoi region** $\mathcal{V}_E$ contains all points closer to E than to any P_i:

$$\mathcal{V}_E = \{x \in R^3 : \|x - E\| \leq \|x - P_i\|, \forall i\}$$

- As long as the evader remains within $\mathcal{V}_E$, it can avoid capture by moving in the direction of any $x \in \mathcal{V}_E$ at speed v_{max}
- The Voronoi region is dynamically updated at each timestep

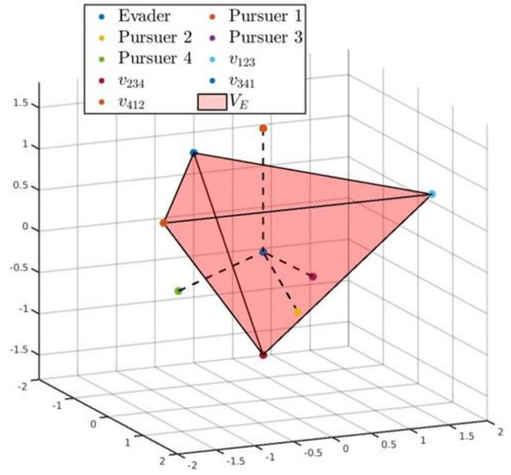
Capture Conditions and Geometry

- In 3D, the Voronoi region of the evader is bounded iff it lies inside the convex hull of at least 4 non-coplanar pursuers
- The evader can escape only if:

$$E \notin \mathrm{ConvHull}(\{P_1, P_2, \ldots, P_n\})$$

- If E lies inside the tetrahedron formed by 4 pursuers, it is guaranteed to be captured eventually

Evader's Voronoi region



Adapted from Adityaya Dhande's BTP Presentation: *A Multi-Pursuer Single-Evader Pursuit-Evasion Game*

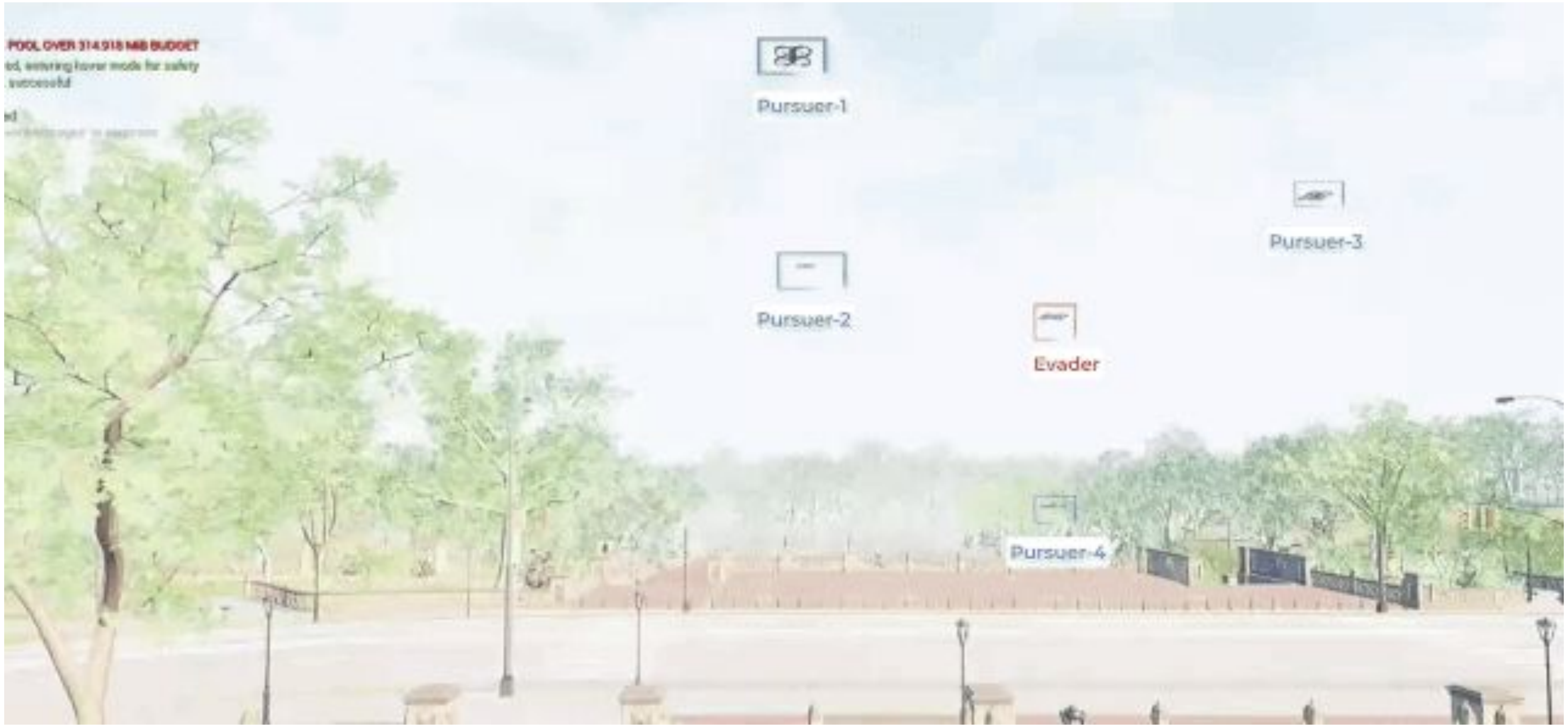
Voronoi Based Pursuit Strategy

- Let $\mathcal{V}_{ijkE}$ be the Voronoi vertex formed by pursuers P_i, P_j, P_k and evader E
- Each pursuer P_s moves toward the Voronoi vertex V_{ijkE} using:

$$\vec{u}_s = -v_{\max} \cdot \frac{P_s - V_{ijkE}}{\|P_s - V_{ijkE}\|}$$

  where $v_{\max}$ is the constant maximum speed of the pursuer
- This control law ensures each pursuer attempts to reduce its distance to the Voronoi vertex in real-time, with normalized direction and maximum speed
- Control inputs are updated in real-time as positions evolve

Voronoi Chase Simulation



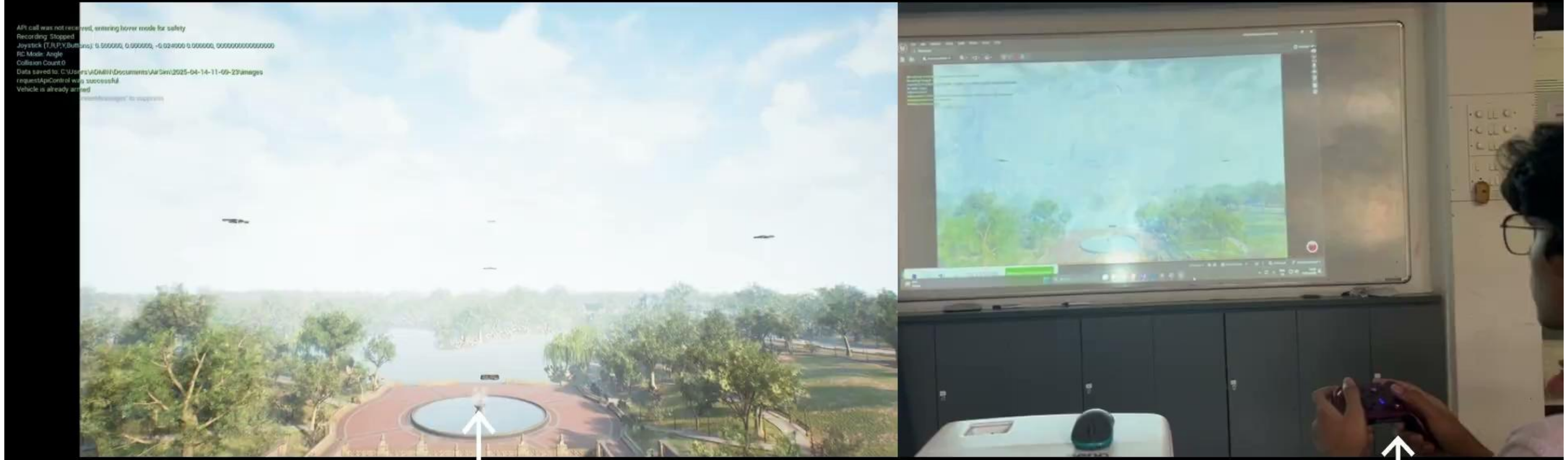
```
POOL OVER 314.918 MiB BUDGET
ed, entering hover mode for safety
 successful

ed
[illegible]
```

Infrastructure Defense System

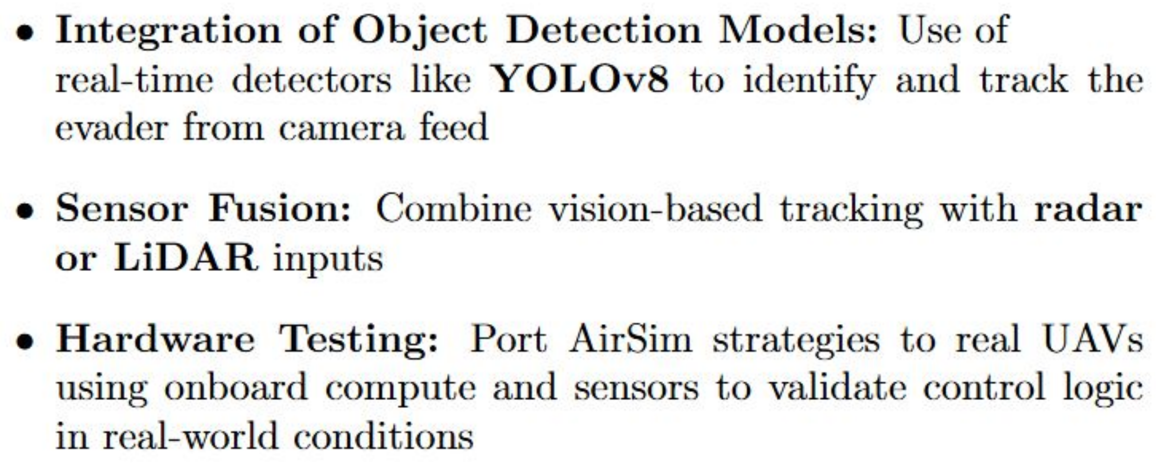
- The objective is to protect a static asset (e.g., a fountain) from an incoming aerial threat
- The Evader drone simulates a hostile UAV attempting to attack the target
- The Evader is **manually controlled** by a Remote Operator using an Xbox controller
- A team of **autonomous Pursuer drones** defends the asset using the **Voronoi-based pursuit strategy**
- The pursuers dynamically coordinate to intercept Evader drone before it reaches the target

Infrastructure Defense Simulation



```
API call was not received, entering hover mode for safety
Recording: Stopped
Joystick (T,R,P,Y,Buttons): 0.500000, 0.000000, -0.024000 0.000000, 0000000000000000
RC Mode: Angle
Collision Count:0
Data saved to: C:\Users\ADMIN\Documents\AirSim\2025-04-14-11-09-23\images
requestApiControl was successful
Vehicle is already armed
```

Future Work



- **Integration of Object Detection Models:** Use of real-time detectors like **YOLOv8** to identify and track the evader from camera feed
- **Sensor Fusion:** Combine vision-based tracking with **radar or LiDAR** inputs
- **Hardware Testing:** Port AirSim strategies to real UAVs using onboard compute and sensors to validate control logic in real-world conditions

Credits and Acknowledgements

Professor Debraj Chakraborty for his continuous guidance and support throughout the project

Voronoi-Based Strategy

- Theory developed by **Deep Boliya** and **Adityaya Dhande**
- Their formulation of the evader's Voronoi region formed the core theoretical basis
- I adapted their ideas into a working simulation strategy

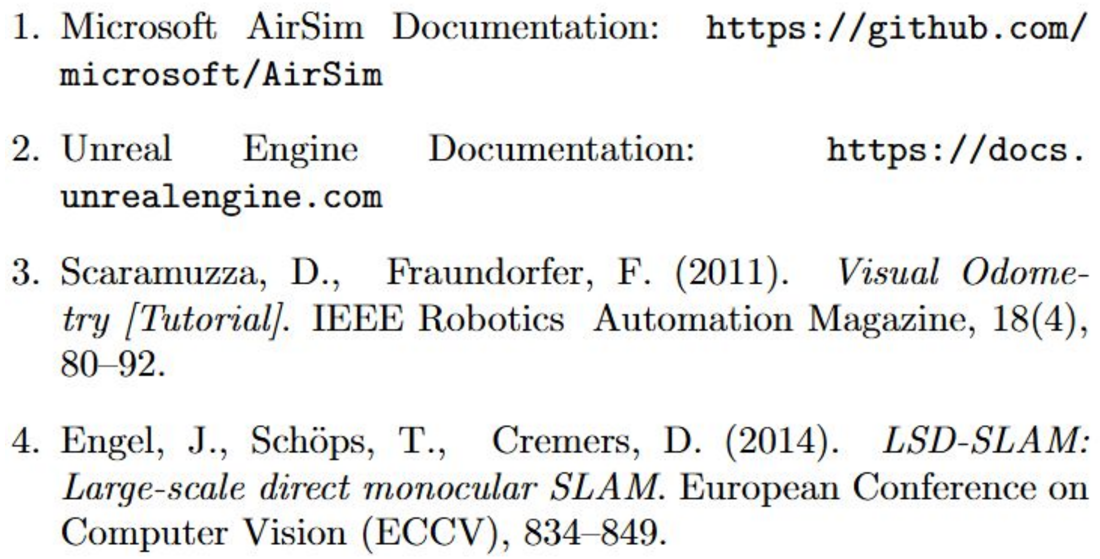
ACGC (Adaptive Camera Guided Control)

- Developed with help from **Anjaneya Damle**
- His insights on geometry and depth modulation were instrumental

Implementation Help

- **Deep** and **Anjaneya** also supported AirSim implementation and debugging

References



1. Microsoft AirSim Documentation: `https://github.com/microsoft/AirSim`
2. Unreal Engine Documentation: `https://docs.unrealengine.com`
3. Scaramuzza, D., Fraundorfer, F. (2011). *Visual Odometry [Tutorial]*. IEEE Robotics Automation Magazine, 18(4), 80–92.
4. Engel, J., Schöps, T., Cremers, D. (2014). *LSD-SLAM: Large-scale direct monocular SLAM*. European Conference on Computer Vision (ECCV), 834–849.

Thank You!

Siddharth Anand | 20D070076