Implementing Safety Mechanisms in Unity with Artificial SLAM and RealSense Location Data

Prasad Prabhu
University of California, Davis

Winter 2026

Abstract

This project develops a Unity-based system for integrating Simultaneous Localization and Mapping (SLAM) pose data into a real-time multi-drone simulation environment, with a primary focus on safety mechanisms. Instead of implementing SLAM itself, this work assumes that SLAM pose estimates are provided externally (e.g., from a computer running SLAM scripts while connected to an Intel RealSense T265) and investigates how such data can be used within Unity to enforce motion-safety constraints. A synthetic SLAM data pipeline was constructed to emulate real-world sensor behavior. In addition to this pipeline, multiple safety layers were implemented, including predictive collision detection, confidence-based motion degradation, degree-of-freedom restrictions, and barrier safety mechanisms. Architecture was developed to support User Datagram Protocol (UDP)-based external SLAM sources and to enable the system's transfer to real-world drone platforms. This project demonstrates how SLAM can be meaningfully leveraged inside Unity to improve safety.

Acknowledgments: I would like to thank Dr. Nelson Max for his expertise and guidance during my project and time in the lab. I also acknowledge that Grammarly and its AI tools were used to improve the writing of this report.

1 Introduction and Background

The main goal of the research lab is to develop a multiplayer augmented reality (AR) drone game in which multiple quadcopters coexist in the same physical and virtual space and compete against each other in real time. Players control each Holybro quadcopter with Oculus Touch controllers, and when combined with a mounted ZED2 stereo video camera, RealSense SLAM camera, onboard computer, and Oculus Rift goggles, the system provides the player with an AR experience to make them feel as if they were looking through the windows of a real aircraft in the actual environment in which they are flying. During this game, additional workstations perform Simultaneous Localization and Mapping (SLAM)

on feature points in the environment to compute the feature points' 3D positions. The output of SLAM is often referred to as 'pose data', which contains information describing the camera's location and orientation in the world. This information is communicated to the master computer for use in game physics calculations. The master computer receives the user's flight-control signals and, before sending them directly to the quadcopters, may modify them according to the game's physics to enforce constraints [1].

With this goal in mind, when I began working on this project in December 2025, several aspects were still under development. Repairs were being made to the drone to return it to its previous functionality, SLAM was being reworked and brought up to date with modern software versions, and new games were being built on the newest versions of the Unity game engine to utilize newer features. With this development underway, my project addresses a critical gap by uniting the different aspects of the project. Once the SLAM systems are complete, they will primarily serve as a localization backend to compute the 3D positions of the drones and align them within a common coordinate system. However, there is limited infrastructure for integrating SLAM outputs with the higher-level Unity system (which can modify the motion of the physical drones to enforce constraints or conditions) to enable the use of localization data during gameplay. This gap becomes even more critical when we consider that multiple drones will share the same constrained airspace. Without that connection between SLAM and Unity, the game engine cannot be sure of the true position of the drones (due to drift between where Unity thinks the drones are and their true physical location), and lacks the situational awareness needed to accurately predict collisions and adapt the drone behavior to maintain safety under various circumstances. This limitation not only affects the quality and enjoyment of gameplay but also poses real-world safety risks, such as mid-air collisions.

Thus, the primary motivation and goal of my project is to address this gap by designing a SLAM-to-Unity framework that takes into account the pose streams of each drone (after SLAM has placed them within the same coordinate system) and incorporates this information into Unity's game logic to support safety mechanisms that make drone operation safer during gameplay.

1.1 Prior Work

Before joining the project, a previous student developed a working implementation of SLAM that placed two drones in the same coordinate system by modifying the RealSense software development kit [2]. However, since then, his implementation has had issues running on our workstations due to the usage of outdated and unsupported dependencies. Other students are currently working in the lab to revise and restore his solution to its original functionality. However, limited work was done to integrate his SLAM solution with the Unity game engine, as only a script was developed to transmit JSON data from his solution to Unity. SLAM pose estimates contain noise due to feature-matching uncertainty and inertial-sensor drift; previous work in this lab addressed this by implementing a moving-average filter to smooth the pose estimates and by incorporating image-preprocessing techniques such as barrel-distortion correction to improve feature-tracking accuracy. Although this project

does not implement these techniques, future work can expand the system to support moving-average filtering or other time-series filtering methods.

1.2 Contributions

My contributions to this project are the following:

1. A synthetic pose generation framework to simulate SLAM output, allowing for SLAM-related features to be tested while the actual implementation is still in development.
2. Pose handling and Unity integration to allow Unity to use data generated by SLAM.
3. A basic pose-quality monitor to characterize communication stability or system degradation.
4. Safety mechanisms that use quality monitoring and pose data to predict and avoid collisions and to constrain motion during periods of system instability or degradation.

1.3 High-Level Overview

At a high level, this project receives time-stamped pose estimates, feeds them into a centralized management script in Unity, and then exposes this information to modules for visualization, diagnostics, and safety control. Design choices were made specifically to separate the synthetic aspects from the Unity-side handling and safety logic. This was done to ensure compatibility with real hardware-generated SLAM outputs. The current synthetic pipeline is shown in Figure 1. Here, two game objects in Unity were treated as "ground-truth," each representing a physical drone, and movement controls move this object. That movement is recorded and sent to a Synthetic SLAM Pose Generator that adds drift, noise, and degradation (for added realism expected in actual pose data) to the movement before generating pose data in a similar format as expected with real SLAM hardware. The pose data is then packaged into a UDP packet and sent to the UDP_PoseReceiver script in Unity. The receiver then forwards the information to the central manager, "SLAM_SystemManager," which is responsible for using the pose data for pose routing to various scripts, velocity estimation, pose quality monitoring, collision prediction, and safety scaling to limit drone movement based on circumstances. In future work using physical SLAM hardware, only minor modifications to this flow are required to achieve the same functionality. Instead of ground-truth drones within Unity, real drones and SLAM cameras send out the SLAM pose estimates; then instead of using a synthetic SLAM pose generator, actual external SLAM processes align drones and package the pose information to be sent to the UDP_PoseReceiver via a UDP pose packet; thus, the rest of the flow and my implementation remain the same.

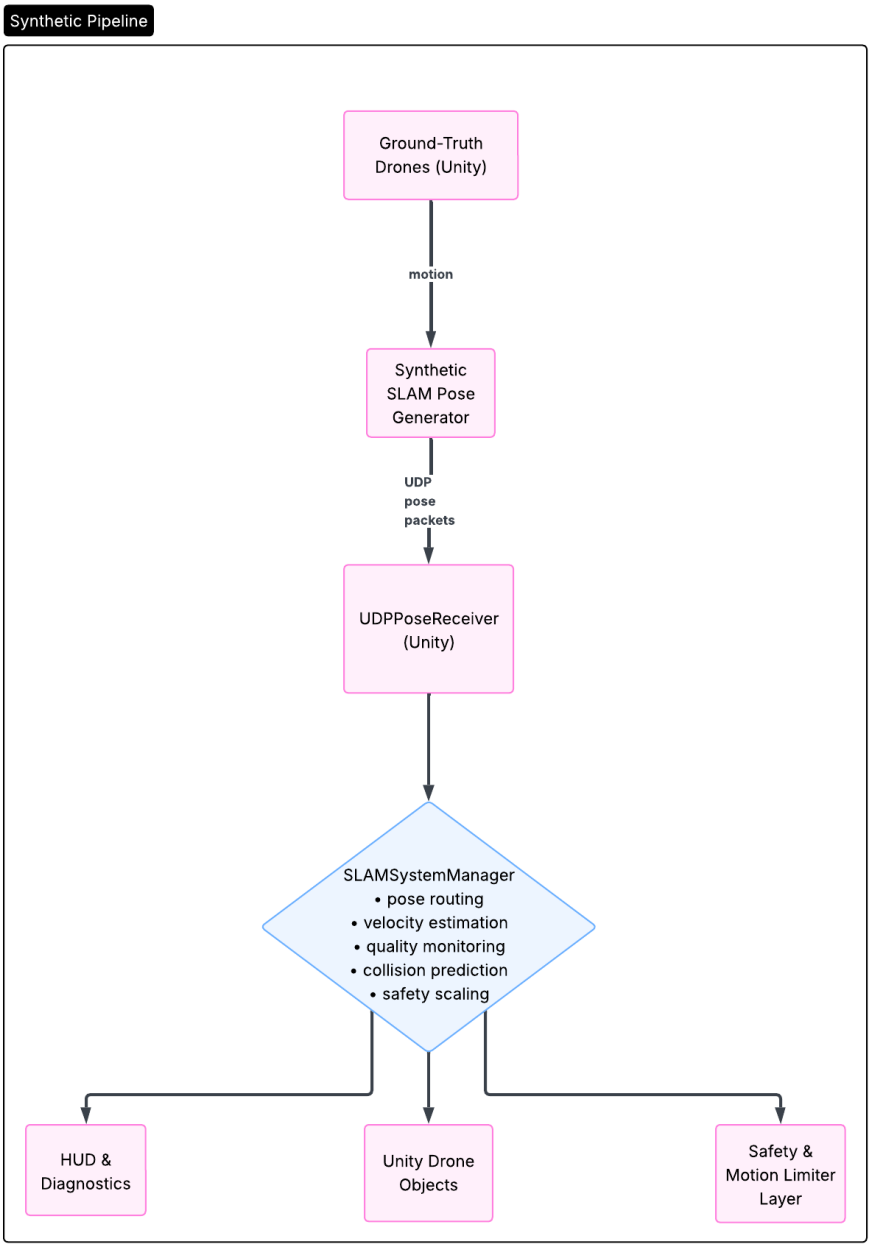

Figure 1: High-Level Project Overview

2 Implementation

2.1 Synthetic SLAM Simulation, UDP Integration, and System Instrumentation

There are two main components to my project. This first section focuses on developing a realistic yet synthetic SLAM pipeline that requires no physical hardware beyond the Unity engine and the workstation. This section details synthetic pose generation, UDP-based pose transport, Unity-side pose consumption, and system instrumentation to support diagnostics. These elements serve as the foundation for the project and provide the structure and information required for the safety mechanisms (see Section 2.2) to operate.

2.1.1 Ground-Truth Abstraction and Virtual Drone Control

Given that the physical drones and SLAM computation remain under development and are not yet functional, I needed to simulate certain aspects of the project to make progress on the SLAM-to-Unity pipeline. The first step I needed to simulate was real-world drones, and to do so, I introduced two ground-truth drone objects in Unity. These objects represented the physical drones and were directly controlled by a custom controller component, allowing control via a keyboard.

These ground-truth objects serve two purposes:

1. They serve as a controllable source of motion.
2. They represent the "physical world" from which SLAM will compute poses.

This was the first step in setting up a synthetic SLAM simulation.

2.1.2 Synthetic SLAM Pose Generation

Given that other students are currently working on the SLAM algorithm and implementation, I did not re-implement a full SLAM pipeline for my project. Instead, I introduced a synthetic pose-generation layer that uses motion from the ground-truth objects in Unity. That ground-truth motion is then converted into a simulated SLAM output that resembles the output (pose data) from a real system. Instead of simply adding a straightforward conversion into SLAM output, additional modifications were made to the data to resemble more realistic behavior, such as noise, positioning drift [3], jitter, and a controller to manually induce degradation and confidence loss (for testing purposes).

Based on movement updates from the ground-truth objects, the system constructs a PoseData packet containing:

- DroneId (unique integer identifier assigned to each drone, similar to the RealSense camera ID if utilizing physical hardware)
- Timestamp

- Position
- Rotation
- TrackingConfidence

These packets are then serialized and broadcast over UDP to simulate an external SLAM system sending UDP pose packets. By structuring the simulated elements this way, the downstream Unity components can maintain their behavior when the simulated SLAM packet flow is replaced with real hardware.

2.1.3 UDP-Based Pose Transport

All pose data are transmitted via UDP to align with the setup of our research lab project. Given that it's unrealistic and problematic to have a hardwired connection from physical drones flying around, an onboard computer sends the pose packets via wireless to the master workstation. Given this setup, the synthetic SLAM provider emits pose packets via UDP, and a dedicated UDP receiver in Unity reconstructs these packets for use within the environment.

2.1.4 Centralized Pose Management

All incoming pose data are processed by a central coordination script implemented in the SLAM_SystemManager software component. Most of the logic in the project is handled by this component, which performs actions such as:

- Receiving and routing pose packets to the relevant helper files/functions.
- Maintaining per-drone state.
- Updating Unity drone representations.
- Estimating velocities from pose data.
- Handling the logic of safety mechanisms.

An important distinction to make is that the "Unity drone representations" are not the same as the ground-truth objects, despite the fact that they are both implemented in Unity for this project. The ground-truth objects represent the physical drones, whereas the Unity drones represent the virtual drones simulated by all our games. Unity drones are never defined directly in my project; instead, they are defined by SLAM-derived pose estimates.

2.1.5 Pose Quality Monitoring and Diagnostics

To implement safety mechanisms that modify behavior based in part on pose quality, I needed a basic diagnostic implementation that analyzed incoming pose packets and recorded metrics. For each drone, I tracked:

- Packets per second.
- Time since last pose update.
- Exponential moving average of frame time (the smoothed average time between pose packets).
- Jitter (how much packet timing deviates from the average).
- SLAM-reported confidence (a reliability estimate associated with each pose that is computed by my synthetic SLAM pose generation software based on factors such as feature tracking quality and map stability).

With every incoming pose packet received, these metrics are computed and updated to provide an estimate of the health and stability of the system.

2.1.6 Visualization and Heads-Up Display

I designed a basic heads-up display (HUD) to make certain metrics more easily observable for debugging purposes. The HUD takes in information from the SLAM manager and quality monitoring systems and presents data such as:

- Per-drone SLAM health (generic categorization of the overall health of the drone based on the incoming SLAM pose diagnostic information).
- Pose freshness and jitter.
- Estimated velocities.
- SLAM confidence level.
- Predicted collision time.
- Collision prevention metrics.

The HUD makes these various aspects of the project interpretable and facilitates both debugging and evaluating the functionality of my approaches. Figure 2 shows the HUD.

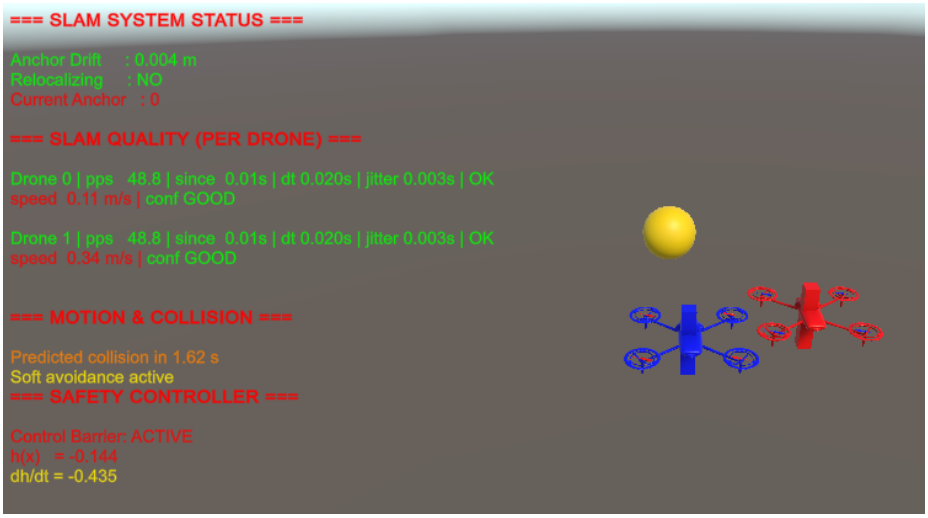

Figure 2: Basic HUD Metrics

2.2 Safety Mechanisms and SLAM-Driven Control

The elements of the previous section serve as the foundation for the project and provide the structure and information necessary for the safety mechanisms of this section to operate. This section focuses on safety mechanisms and the modification of drone behavior based on SLAM pose data. By analyzing and interpreting each drone's SLAM state, safety mechanisms can constrain drone motion during pose uncertainty, enable predictive collision detection, and implement preemptive slowdowns and barrier safety to prevent full collisions. Most of the mechanisms listed below are handled by the central loop of SLAM_SystemManager.

2.2.1 Velocity Estimation from SLAM Poses

To make drone operation safer, the ability to control and limit drone speed is critical. To do this, we first need the drone's velocity before applying these limitations. Normally, our quadcopter has an onboard inertial measurement unit (IMU) that provides a means of estimating the drone's velocity by combining IMU data with SLAM data using Kalman filtering to obtain accurate and stable velocity estimates. Since the physical drone IMU wasn't available during this project, the velocity was instead reconstructed from pose differences; while this approach is more sensitive to noise than Kalman filtering, it is sufficient for collision prediction, and future work can integrate Kalman filtering into the system to improve velocity estimation. Velocity is reconstructed from successive pose packets by the formula:

$$v_i(t) \approx \frac{p_i(t) - p_i(t - \Delta t)}{\Delta t}$$

where $p_i(t)$ is the current position of the drone i, and Δt is the packet time interval. In the code, this formula is implemented by the lines:

```
Vector3 velocity = (currentPos - prev.position) / (float)dt;
_estimatedVelocity[droneId] = velocity;
```

2.2.2 Core Monitoring and Safety

One of the core safety features is to limit drone movement and autonomy whenever performance degrades. This is because if data sent to Unity is corrupted or lost, Unity cannot be certain of the drones' true positions, and any safety mechanisms are no longer reliable enough to prevent collisions. If we aim for absolute safety, we could completely restrict drone movement during periods of degradation; however, in practice, extended periods of degradation are unlikely and brief periods are more common. To that end, instead of fully locking movement, which could disrupt gameplay if triggered frequently and briefly, I opted for an alternative method of limiting degrees of freedom (DOF) based on SLAM confidence and system performance.

Given that I previously implemented packet monitoring and diagnostic information, I could rely on metrics such as packet arrival rate, time since last packet, jitter, and other metrics as triggers. I first handled the case of an extended period of degradation. Although this is unlikely, it indicated a critical issue and required full lockdown of drone motion. If no pose packet is received within a configurable threshold (i.e., time since the last packet exceeds the staleness threshold), the drone is flagged as stale, and updates to its motion are frozen, meaning that the drone's commanded velocity is set to zero (coming to a complete stop), and its position remains constant until valid pose updates resume. Essentially, this trigger is activated when no pose packets have arrived within a certain interval.

The more common scenario is an occasional performance drop, in which certain DOF are limited by the confidence of the pose packets. This feature relies on the assumption that the SLAM implementation synthesizes its own confidence metric from several uncertainty signals and fields recorded by the SLAM camera, and that it includes this confidence with each pose packet. In my implementation, I assumed confidence values within the range {Good, Degraded, Poor, Lost}; however, these metrics and implementation can be tailored to the specific SLAM pose packets. Based on this metric, the following constraints were applied as shown in Table 1. This is implemented by applying an axis mask to the motion controller to enforce the constraint that, as SLAM quality decreases, the system starts applying stricter limitations to the drone movement.

2.2.3 Predictive Collision Modeling

In game development, collisions can be handled by assigning bounding spheres to objects that define the regions of space they occupy. The intersections of these bounding spheres indicate a collision, and constraints can be applied to limit further movement. However, given that we have real-world elements (the drones), this method is insufficient because the drones cannot immediately stop their velocity like you can in a game; thus, a predictive

Confidence	Permitted Motion
Good	Full 3-DOF Translation + Yaw
Degraded	2D Translation Only (Vertical Locked) + Yaw
Poor	Yaw Only (No Translation)

Table 1: DOF Contraints

element is needed to identify a collision before it occurs so that there is sufficient time in the real world to prevent it. To implement this predictive measure, I rely on relative kinematics, and for each drone pair (i, j), I compute:

Relative Position: $p_{rel} = p_j - p_i$

Relative Velocity: $v_{rel} = v_j - v_i$

Time of closest approach:

$$t' = -\frac{p_{rel} \cdot v_{rel}}{||v_{rel}||^2}$$

Predicted separation at closest approach: $||(p_i + v_i t') - (p_j + v_j t')||$

Using these formulas, I can predict relative-motion collisions by computing these values at each game tick for all drone pairs.

2.2.4 Continuous Slowdown Safety Constraints

With the predictive element in hand, I was then able to provide safety constraints. I applied two different collision avoidance mechanisms, the first being continuous slowdown as described in this section, and the second being a barrier safety mechanism described in the next section. First, I proposed a nonlinear slowdown mechanism that causes drones to gradually slow down when their motion predicts a collision, with a slight decrease in speed at a larger range and a much stronger slowdown at a closer range.

Let d denote the predicted closest separation distance between two drones. Two distance thresholds are:

- d_{slow}: outer slowdown radius, where speed reduction begins
- d_{hard} inner slowdown radius, where maximum slowdown occurs

with $d_{hard} < d_{slow}$.

The slowdown scaling function is defined as

$$s_d(d) = \begin{cases} 0 & d \le d_{\text{hard}} \\ \left(\dfrac{d - d_{\text{hard}}}{d_{\text{slow}} - d_{\text{hard}}} \right)^4 & d_{\text{hard}} < d < d_{\text{slow}} \\ 1 & d \ge d_{\text{slow}} \end{cases} \tag{1}$$

which acts as a multiplier of the proposed velocity to provide a smooth, nonlinear deceleration as the drone approaches the inner slowdown radius. The fourth-power curve was selected to achieve stronger deceleration near the inner boundary without significantly affecting the smoothness of the motion at larger distances.

In Unity, this was implemented by the lines:

```
float t = Mathf.InverseLerp(hardStopDistance, slowDownDistance, distance);
float curved = Mathf.Pow(t, 4f);
```

Relying on distance alone is insufficient to ensure safety. The above slowdown effects may be enough to prevent a collision between two slow-moving drones, but they may not be enough to stop two drones rapidly approaching each other at speed. Therefore, braking is also regulated by the speed of the drones.

Let $\hat{p}_{rel} = \frac{p_{rel}}{||p_{rel}||}$ denote relative direction. The closing speed is computed as $v_{closing} = max(0, v_{rel} \cdot \hat{p}_{rel})$

The velocity scaling function is defined as:

$$s_v = lerp(1, s_{min}, \frac{v_{closing}}{v_{agressive}})$$

where $v_{aggressive}$ is a tunable velocity threshold, s_{min} is the minimum speed scaling factor, and lerp is linear interpolation. $v_{aggressive}$ and s_{min} were arbitrarily chosen based on the motion observed in the Unity simulation. In the real quadcopter system, these parameters should be derived from physical deceleration limits and system latency.

The slowdown and velocity scaling functions act as a multiplier on the proposed velocity from the user to give the final command velocity sent to the drone as shown in

$$s_{final} = s_d(d) * s_v * s_{proposed}$$

2.2.5 Barrier Safety (Repulsive Field)

The second implemented collision avoidance mechanism was barrier safety. I decided to pursue an alternative method to move drones away from each other as opposed to completely stopping drones within close proximity to each other (where slight drone drifts may then still result in a collision even if no movement controls were applied). I decided to implement a barrier safety approach, which acts as a repulsive field to redirect drones in close proximity and was inspired by the Control Barrier Function theory as described in [4].

Let the safety function be defined as

$$h(p) = ||p_{rel}||^2 - d_{min}^2$$

where p_{rel} is the relative position and d_{min} is the minimum safety distance. The derivative of the safety function with respect to time is: $\dot{h}(p) = 2p_{rel} \cdot v_{rel}$

Safety is enforced using the barrier constraint:

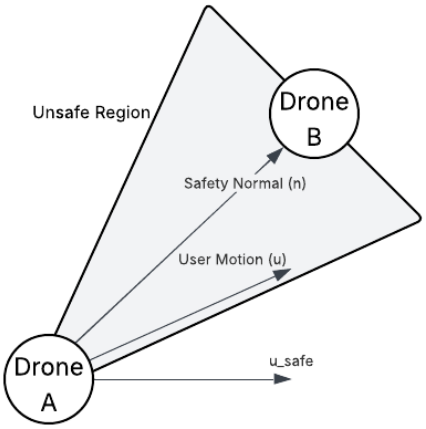

Figure 3: Directional Motion Projection

$$\dot{h}(p) \geq -\alpha h(p)$$

where α is a tunable parameter to adjust the responsiveness. This means that a violation occurs when $\dot{h}(p) < -\alpha h(p)$, and this condition predicts when the system is entering an unsafe region.

When a violation is detected, the system does not completely stop the drone movement. Instead, it removes only the unsafe component of motion. Let u be the user motion command and let the safety normal, n, be

$$n = \frac{p_{rel}}{||p_{rel}||}$$

If the user motion is deemed unsafe by satisfying: $u \cdot n > 0$, then the unsafe component is removed to attain the safe motion:

$$u_{safe} = u - (u \cdot n)n$$

which allows motion that avoids or separates the drones, but prevents motion that moves them closer together toward the unsafe region. See Figure 3.

In Unity:

```
float toward = Vector3.Dot(movement, rejection.worldNormal);
if (toward > 0f)
{
```

```
    Vector3 blocked = toward * rejection.worldNormal;
    movement -= blocked;
}
```

The resulting behavior of this barrier safety mechanism in the context of our system is that it creates a repulsive safety field with the following properties:

- Motion toward unsafe regions is prevented
- Motion away from unsafe regions is preserved
- Tangential escape motion is preserved
- Drones naturally deflect away from hazards

Barrier safety primarily redirects drones and serves as an alternative to the continuous slowdown avoidance mechanism described in Section 2.2.4, which focuses on limiting drone velocity. With careful tuning of the parameters for each method, both safety mechanisms can also operate simultaneously. Continuous slowdown provides smooth braking and early collision intervention, while barrier safety can redirect the drone to a safer region if the slowdown constraints are insufficiently tuned to prevent a collision at high velocity.

Both mechanisms were used effectively at the same time in the Unity environment by configuring the hard-stop radius of continuous slowdown to be smaller than the barrier activation radius, $d_{hard} < d_{barrier}$. This ensures that barrier safety is activated before the drone reaches a full stop. The combined approach yields smooth motion at long distances, strong braking at medium distances, and repulsive collision prevention at close range.

2.2.6 Environmental Safety Mechanisms

The safety mechanisms discussed so far were designed primarily for interactions between two drones. The system was later extended to apply these mechanisms to static environmental objects such as walls, floors, ceilings, obstacles, and other structures. This functionality is implemented through the SafetyObstacle component in Unity, which registers these objects with the SLAM_SystemManager so that similar safety calculations can be performed as described in previous sections. However, automatically defining appropriate safety regions for environmental objects of varying shapes and sizes remains an area for future improvement.

2.2.7 Indirect User Operation

These safety mechanisms were designed to operate in tandem with the existing indirect user-operational constraints applied in our Unity games. Previous work has established a pipeline in which user input for drone movement first passes through Unity before commands are sent to the virtual/physical drones. The drones are never controlled directly by the user. My implementation aligns with this approach, as movement goes through the

following flow to enforce the safety mechanisms implemented:

User Input ⟶ Confidence Based DOF Constraints (Section 2.2.2) ⟶ Predictive Safety Slowdown (Section 2.2.4) / Barrier Safety (Section 2.2.5) ⟶ Executable Motion

The user does not directly control the drone; instead, the system controls the drone's motion to approximate the user's intended trajectory while accounting for safety constraints. Note that Predictive Safety Slowdown and Barrier Safety mechanisms can be used at the same time, but require careful consideration of the settings and parameters to avoid conflicting behavior.

3 Methodology and Setup

3.1 Architecture

In terms of architecture, the only requirement for running this project is a Windows workstation that meets Unity's minimum system requirements.

3.2 Project Setup

To set up and run the project for testing, all that is needed is to download and run the code written in the project repository [5]. The readme file will go into more detail on how to test the project.

4 Results

The results of the safety mechanisms were showcased in the video linked in the references [6]. This section serves as a narration describing the behavior seen in the video. Note that most of the safety mechanisms have tunable parameters, so aspects such as the intensity of slowdown, radius of the repulser field, and DOF constraints can all be modified to fit the required behavior of a real implementation, and the parameters set during this demo may not reflect the real parameters used in a live test with physical quadcopters.

4.1 Video Description

At the start of the scene, there were two different-colored drones and vertical columns near the center of these drones. These columns represented the ground-truth, and I created a controller that allowed me to move them. I also had Unity drones that corresponded to where SLAM estimated the drones were and were positioned based on the SLAM pose data.

The video began with basic movement of the drone. The blue drone (drone 0) was then moved in all directions and rotated. Most of the movement was "safe," so the HUD

reported speeds exceeding 2 m/s at times, its highest speed. Note that even when both drones received no player input, their speed still fluctuated slightly. This occurred because the speed was estimated based on the differences between successive pose packets and because the generated pose packets also had drifting measurements.

The next test demonstrated one of the core safety features. I disabled the script that provided the pose, simulating a catastrophic communication failure for drone 0. Immediately, that drone was marked as stale, and the time-since-last-packet-arrival metric (since) began counting upward. In this case, the drone's motion was frozen until reconnection, but drone 1, without issue, continued to function normally. Next, I tested the DOF limitations. I enabled a script to trigger performance degradation for testing purposes, and the confidence of drone 1 in the HUD was marked as limited. Immediately, the drone speed was reduced and only reached a peak of over 1 m/s. In this state, the DOF limitation was that vertical movement was blocked; even when I pressed the inputs, the drone did not move vertically. The next limitation was when the drone was in a poor confidence state. In this state, the drone could only rotate, and all other movement inputs were blocked.

The next test demonstrated predictive collision and continuous slowdown. The metrics under the "== MOTION & COLLISION ==" section in the HUD indicated predicted collisions, and if a collision was predicted, a slowdown was occurring at the same time. As the drone approached another drone, its speed decreased, dropping to approximately 0.23 m/s at times. In this demo, a slight slowdown was used; however, the slowdown could become more drastic. Distance between the drones wasn't the only metric triggering the slowdown and collision prediction; the drones' speed was also taken into account. Another test was performed in which both drones approached each other rather than one remaining motionless, and the slowdown was triggered earlier and began at a greater distance.

The final test demonstrated the repulsive safety-barrier control mechanism. The blue drone was continuously moved toward the red drone; when it approached a specified range, it was redirected to glide around the red drone. Note that for the user input, I only pressed the movement keys in a single direction towards the drone and no other inputs were pressed; for example, in the first test, the blue drone was on the left and moving towards the red drone on the right, and I only held the input key for moving the drone to the right; then as the drones approached, the blue drone was forced to move under the red drone to prevent a direct collision, but I only held the right movement key.

5 Future Work

Future work involves integrating this project with real hardware rather than relying solely on simulated hardware. Once this integration is completed, this project will make a valuable contribution to drone operations and help bridge the gap between Unity and SLAM.

6 Conclusion

This project presents a practical framework for integrating Unity and SLAM to provide safety mechanisms in a multi-drone environment. The system enables Unity to consume pose, velocity, and quality information via synthetic SLAM generation, UDP-based pose packets, and quality monitoring, and to implement a safety architecture that includes features such as predictive collision analysis, confidence-based motion scaling, and DOF constraints. The design of this project was aimed at maintaining extensibility for real hardware input and making drone operation safer and easier for players.

References

[1] "Augmented Reality Quadcopter Project." [Online]. Available: https://cs.ucdavis.edu/graduate/current-students/available-research-projects

[2] "quadcopter-ar/librealsense," Nov. 2025, original-date: 2023-04-14T00:24:16Z. [Online]. Available: https://github.com/quadcopter-ar/librealsense

[3] J. C. K. Chow, "Drift-Free Indoor Navigation Using Simultaneous Localization and Mapping of the Ambient Heterogeneous Magnetic Field," Feb. 2018, arXiv:1802.06199. [Online]. Available: http://arxiv.org/abs/1802.06199

[4] A. D. Ames, S. Coogan, M. Egerstedt, G. Notomista, K. Sreenath, and P. Tabuada, "Control barrier functions: Theory and applications," in *2019 18th European control conference (ECC)*. Ieee, 2019, pp. 3420–3431.

[5] pprabhu2727, "pprabhu2727/SLAM-Unity-Integration," Feb. 2026, original-date: 2025-12-08T22:34:38Z. [Online]. Available: https://github.com/pprabhu2727/SLAM-Unity-Integration

[6] "Demo.mp4." [Online]. Available: https://drive.google.com/file/d/1RxQbn7q4cQ-ZbqJAaoyM3-HlEL1WAcg4/view?usp=sharing&usp=embed_facebook