Citation: David Peterson, Beyonce Andrades, Kevin Lizarazu-Ampuero, Jai Deshmukh, Thomas Stapor, Will Destaffan, Don Engel, Justin Krometis and Justin A. Kauffman. "Integration of Reinforcement Learning and Unreal Engine for Enemy Containment via Autonomous Swarms," AIAA 2023-2674. AIAA SCITECH 2023 Forum. January 2023. https://doi.org/10.2514/6.2023-2674

DOI: https://doi.org/10.2514/6.2023-2674

Integration of Reinforcement Learning and Unreal Engine for Enemy Containment via Autonomous Swarms

David Peterson, Beyonce Andrades, Kevin Lizarazu-Ampuero, Jai Deshmukh
Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24061

Thomas Stapor
Department of Computer Science, Virginia Tech, Blacksburg, VA 24061

Will Destaffan, Don Engel
Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Baltimore, MD 21250

Justin Krometis
National Security Institute, Virginia Tech, Blacksburg, VA 24060

Justin Kauffman
National Security Institute, Virginia Tech, Arlington, VA 22203

Maritime remote sensing (MRS) is a multi-disciplinary and multi-physics field at the intersection of naval hydrodynamics, physical oceanography, overhead platforms, and electro-optical sensors. One proposed improvement to MRS information gathering and operations is the use of swarms of autonomous surface, aerial, and/or undersea vehicles as a multi-agent system (MAS) to automate data collection, data processing, and situational awareness.

Here, we explore the design of an autonomous multi-agent system with the objective of containing a target object, i.e., surrounding the object in a loosely defined shape. The agents make decisions using reinforcement learning by way of a Markov decision process. Our current proof-of-concepts are modeled using Python-based 2D simulation environments which contain our agents and target used for prototyping and testing various reward functions.. However, we have built an infrastructure to port the simulation environments to Unreal Engine 4 for increased fidelity. In the current modeled scenario, each agent's decisions are based on global positional knowledge of each entity in the environment. Future iterations are planned to feature agent decision making based on a high-fidelity communication protocol and inputs from integrated sensors.

I. Introduction

Reinforcement learning is a machine learning technique based on the idea of encouraging desired behaviors by an agent and punishing undesired behavior. This is accomplished by modeling an agent's behavior with a Markov decision process whose parameters are refined by a feedback loop (Figure 1) with a defined reward function. Agents observe

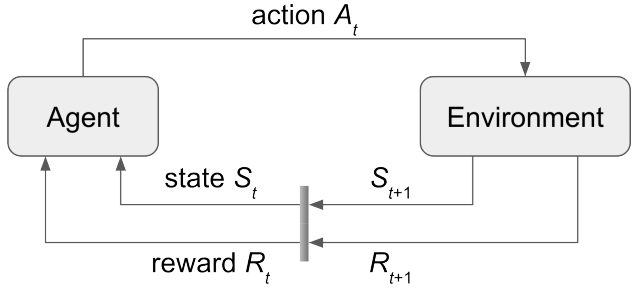


Fig. 1 Flow of Markov decision process used in reinforcement learning model.

their environment, perform an action, and receive a reward based on the outcome of that action. This process repeats for some set time until the agents have completed their goal or have failed to reach their goal by the set time. In our simulation, we define the observable information, possible actions, and reward function of each agent. However, it is possible for reward functions, and even actions, to be learned by the agents themselves. Agents can be made aware of information in the current state of their environment; the possible information the agent can be made aware of is the "observation space" of the agent. In reality, agents can collect data through different types of sensors. This data is what the agent observes and uses in determination of its next action. All the actions the agent can perform are collected in the action space; if an agent can move, then its action space will include the types of movement it is able to perform. The simulation the agent is in is dynamic and can respond to the actions taken by an agent. A simulation primarily does this by providing a reward to the agent, which can steer the agent toward performing certain actions that will lead it to completing its mission. The reward, alongside what the agent observes, drives the behavior of the agent at future time steps in the environment.

A multi-agent system (MAS) is a computerized system composed of multiple intelligent agents interacting with each other to solve complex problems. An intelligent agent is an autonomous entity which acts upon an environment using sensors and collected data. Some examples of an intelligent agent in a MAS are: a wireless sensor for building security [1], internet-of-things connected sensors to detect parking spots in densely populated urban cities [2], and even surveillance agents monitoring different environments [3]. Multi-agent reinforcement learning (MARL) is reinforcement learning which involves a set of agents (e.g., drones or autonomous vehicles). Most use cases for MARL involve the agents interacting and collaborating to achieve a shared objective, such as goals which are too large for a single, centralized agent. Agents perform their behaviors individually, and are locally rewarded based on what they observe and then act upon. Utilization of more than one agent in practical applications increases behavioral coverage and robustness of the swarm of agents. Should one agent malfunction or be unable to perform the desired tasks, the impact on the mission is minimal. As in nature (e.g., flocking [4]), MARL systems can sometimes lead to emergent patterns in behavior within the group.

Inspired by problems in maritime remote sensing (MRS), we focus here on the use of a MARL system where a set of homogeneous drones are tasked with following a target and maximizing their sensors' coverage of this target. The drones are presumed to have an optimum distance from the target which would, for example, prevent them from being detected or attacked while also containing the object within the view of the entire swarm. A 2D simulation environment containing our agents and target is used for prototyping and testing various reward functions. As the long-term goal for the simulation, we model the environment, agents, and target in 3D using Unreal Engine 4 [5] in a flat environment (Figures 2,3) designed with multi-domain maritime use cases in mind.

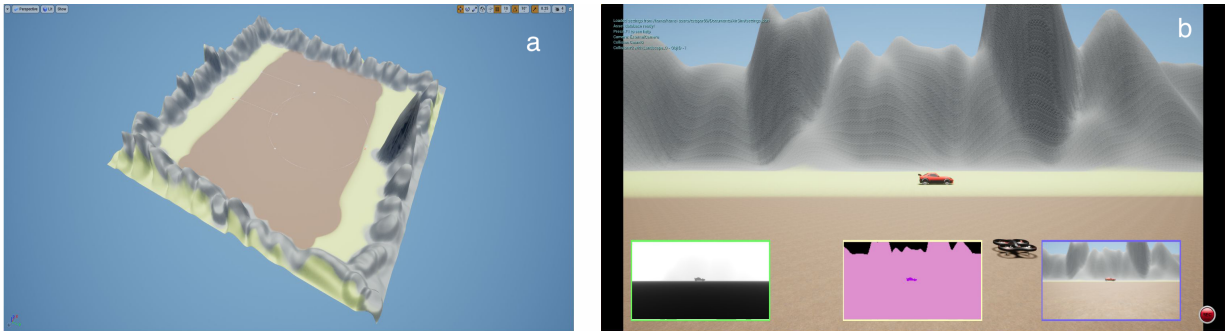


Fig. 2 (a) Overhead view of our Unreal Engine environment. (b) Single target in view of a drone. Types of vision at the bottom of (b), from left to right: Depth, Segmentation, Optical Camera.

Game engines are a powerful tool that can be used to model physical systems. Some common game engine examples are Unity and Unreal Engine. They are used for creating video games as well as physical modeling. Using a game engine, we can build our simulation to behave in a simulated maritime environment with the ability to visualize our agents in real time. Unreal Engine 4 is the second most recent release of Unreal Engine. It is stable and has a multitude of tutorials and resources for creation in the engine. We will be using Unreal Engine 4 to build and ultimately run our simulation.

Unreal Engine has several useful plugins which customize the game engine for specific contexts. Microsoft's AirSim [6] plugin has capability to model aerial drones in Unreal Engine. AirSim models the physical properties like

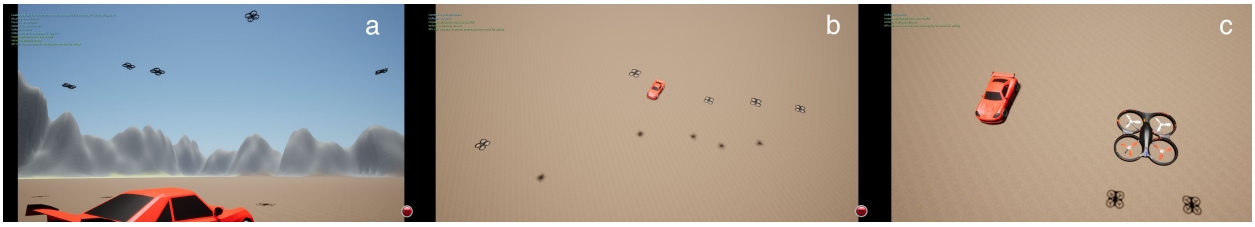


Fig. 3 (a) Target perspective of drones. (b) Drone perspective of a target. (c) Close up of drone over target.

the drone's velocity or acceleration and reflects these constraints in the engine. The drones also come with different sensors that have a view of the game space from the perspective of a particular drone.

Overall, this paper is a summary of an ongoing effort towards building out the MARL framework described above in Unreal Engine by integrating information from the fields of reinforcement learning, autonomous swarms, gaming engines, and communications. The following sections detail the sources we used as inspiration and motivation followed by descriptions of each component of the ongoing efforts of the overall project.

II. Related Work

We take inspiration from several research efforts into the different components of a total MARL system, namely reinforcement learning, computer vision, reward function design, impact of physical constraints on a MARL system, and the dynamics of swarm behavior. A primary component of a MARL system is the design of the reward function for the agents. Work done in [7] discussed several of the design considerations that go into creating an effective reward function. Their analysis of a reward function working in a multi-agent context provides insight into the design of our own reward function. In recent decades, the improvement of object detection and object tracking using cameras and algorithms has inspired extensive research on reinforcement learning in computer vision. Much of of this work is categorized in [8], which is a survey of reinforcement learning in computer vision applications. Our long-term goal is to have our agents have sensors that they will see the world through, performing actions based on what they see. Understanding the nuances and practical applications of computer vision is vital to building well-behaved agents in our simulation. Several of the papers referenced in this reinforcement learning survey provided us with background and useful information and studies to develop our own computer vision approach.

Several examples of multi-agent systems exist outside of computers and software implementations. Biological multi-agent systems consider humans, animals, and ecosystems as intelligent agents acting upon their environment. MAS's also have myriad uses in transportation, logistics, graphics, manufacturing, power systems, smart grids, and geographic information systems (GIS). To focus on our domain of interest, maritime remote sensing, multi-agent systems could be used to detect and contain oil spills [9] and in applications of maritime security [10]. Despite these being just a few examples of how MAS could be used in MRS, they are exemplars of the broader set of applications which demand the containment of a mobile target. Several parallels can be drawn from different maritime applications as they share similar physical conditions (the maritime environment). Studies have been done [11] explaining that the inclusion of the kinematics of moving agents and adding physical obstacles impeding the agents to increase the complexity of the Pursuit Evasion [11] game. Yet this added complexity is key in maintaining environment fidelity.

The most significant challenge we've tackled in this paper is the merging of our reinforcement learning work into the Unreal Engine simulation. Recent work has been done [12] addressing the lack of simulation tools for visualizing reinforcement learning. Game engines serve as an ideal physical representation that can simulate across several degrees of fidelity and environmental context.

III. Methodology and Implementation

There is a fundamental relationship between remote sensing and multi-agent systems when integrating information from multiple sensors/agents. The application in any environment is similar and the underlying algorithms developed are overlapping. Our primary goal is to develop these applications for a maritime environment and develop applications for MAS to be used for maritime security, but first we choose to simplify our environment and focus on the learning of the agents and the sensor models associated with those agents.

The work so far has been two-pronged. One team has been developing a reinforcement learning (RL) framework to

teach a collection of agents to track and contain a prey. A second team has been developing a simulation environment in Unreal Engine into which the RL framework will eventually be inserted to demonstrate "real-world" fidelity of a multi-agent system with remote sensing capabilities. More details on our efforts to merge the RL work into the Unreal Engine simulation is discussed in Section III.D

A. Reinforcement Learning (RL) Framework

The student team working on the RL aspect of the project have developed a framework using the OpenAI Gym* and RLlib† packages to implement multi-agent prey tracking under the basic assumption that the agents are aware of the position of the prey.

The basic components of the RL framework are:

- **State Space**: The state space consists of the position and velocity of each agent and of the prey (and even multiple prey, see Section III.B).
- **Observations**: Each agent keeps track of the position and velocity of itself, the other agents, and the prey.
- **Action Space**: Each agent can choose an acceleration up to a maximum value in each component.
- **Model**: Agent acceleration is applied, in combination with a deceleration constant representing drag, to update each agent's velocity and then position.
- **Reward**: The reward function is described in Section III.A.1.
- **Policy**: The policy is a six-layer fully-connected neural network trained online as the system collects data.

The choices of state, action, and model are patterned after the Boids model of distributed behavior [13], which provides simple rules that have been shown to produce emergent behavior reminiscent of bird flocks.

The following subsections will elaborate more on our approach towards creating a reward function that promotes containment behavior and future work associated with improving upon the quality and complexity of our work.

1. Reward Function

Our reward function has had the simultaneous goal of (1) rewarding agents for maintaining an optimum distance from the prey and (2) maintaining maximum coverage around the prey by keeping the agents from clustering. To these ends, several functional forms were considered. Table 1 highlights the functions that were initially considered, while also highlighting where each was sub-optimal. **Negative Euclidean distance** achieves the goal of convergence around

Table 1 Initial functional forms of the reward function that were considered. R is a model of a reward, x_i and x_j are the positions of agent i or j, respectively, and r is the radius indicating optimal spacing between the prey and the agents.

Reward Function	Equation	Possible Drawbacks
Negative Euclidean Distance	$R = -\|x_i - x_j\|$	No steep increase in reward as proximity increases
Reciprocal Squared	$R = \frac{1}{\left(\|x_i - x_j\| - r\right)^2}$	Non-ideal asymptotic reward, outshines clustering reward
Gaussian Bell Curve	$R = e^{-\left(\|x_i - x_j\| - r\right)^2}$	Peaks at optimum distance r for agents and prey

a prey, but does not provide a steep increase in reward as the proximity to the prey increases. **Reciprocal squared** achieves convergence, but having an asymptotically infinite reward at zero distance from the prey causes the prey reward to overwhelm the anti-clustering reward. **Gaussian Bell curves** avoid this problem by having a finite peak.

Through experimentation, we arrived at the following equations for the prey-seeking and cluster-minimizing components of the reward function. By adding one to the denominator of a reciprocal squared function, we maintain the simplicity of a polynomial while achieving the key benefits of a Gaussian:

$$\rho_{\text{agent}\to\text{prey}} = \frac{\max_p}{1 + \beta\left(\|x_{\text{agent}} - x_{\text{prey}}\| - r\right)^2}, \tag{1}$$

*https://github.com/openai/gym

†https://www.ray.io/rllib

$$\rho_{\text{agent}\to\text{agents}} = \frac{1}{n}\sum_{i=1}^{n}\left[\frac{\max_f}{1+\beta\left(\|x_{\text{agent}} - x_i\| - r_{\text{agent}\to\text{agents}}\right)^2}\right], \quad r_{\text{agent}\to\text{agents}} := 2r\cos\left(90 - \frac{180}{n}\right) \tag{2}$$

$$R = \rho_{\text{agent}\to\text{agents}} + \rho_{\text{agent}\to\text{prey}}, \tag{3}$$

where ρ is a reward component, $\max_p$ refers to the maximum allowable reward parameter for the distance between an agent and prey, similarly, $\max_f$ refers to the maximum reward parameter allowable between agents' distance to one another, n is the number of agents in the environment, β is the steepness coefficient for rewards which can be fine-tuned, x_{prey} is the location of the prey, and all other parameters are defined in caption to Table 1. The sum of Equations (1) and (2) give the total reward for an agent at an epoch in the simulation, Equation (3). To provide some clarity on the two reward functions created, Equation 1 is maximized when an agent is distance r from the prey, rewarding agents for achieving the right distance from the prey, and minimized when the agents are far from the prey, encouraging the agents to get close to the prey. Similarly, the reward function in Equation 2 encourages agents to achieve a distance of $r_{\text{agent}\to\text{agents}}$ between each other, where $r_{\text{agent}\to\text{agents}}$ is the length of a regular polygon (see Figure 4) with nodes a distance r from the center. The total reward function Equation (3) is therefore maximized when agents form a regular polygon around the prey.

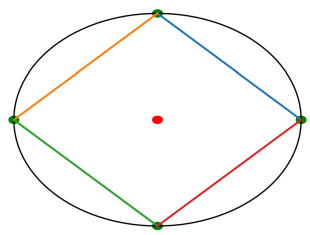


Fig. 4 Optimal agent (green dots) spacing from each other and prey (centralized red dot).

Using the reward functions that were just described as our metric for both training and testing in a simulation with four agents and one prey, the framework above yields increasingly better performance over iterations of reinforcement learning, as shown in Figure 5 – that is, we see the agents increasingly getting closer to the prey and therefore achieving high rewards. However, the reward function also repeatedly drops back down near zero, showing that the agents are allowing the prey to escape the desired distance. Currently, prey movement is random, which means that while the prey is moving around the environment on its own, the actions that it takes are not adversarial whatsoever to the agents in the environment. That being said, more needs to be done to improve swarm containment and how our reward function incentives that behavior. Ideas for improving the RL framework are described in the next subsection III.A.2.

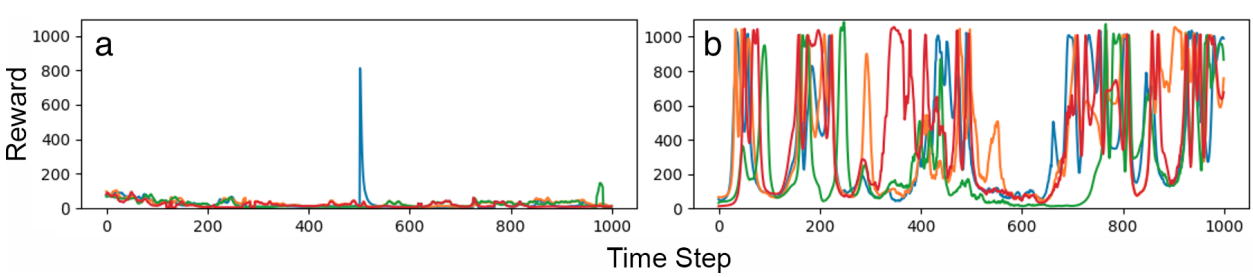


Fig. 5 Reward at each epoch of the simulation for each agent, with each of four agents represented by a distinct color. (a) After first iteration of reinforcement learning. (b) After 5,000th iteration of reinforcement learning. Here $\max_p$ was set to 1000, $\max_f$ was set to 0, β was set to 0.0005, r was set to 30.

Efforts have been dedicated to refining these initial reward functions to obtain faster convergence on our expected behavior and better defined heuristics for which rewards are dispersed. In addition to the standing reward function for anti-clustering, a simpler approach has been explored regarding the agent's Euclidean distance from one another $\rho_{\text{agent}\to\text{agents}}$ in which the reward disbursement mimics the shape of a **Logarithmic curve**. A fixed negative reward is given when the agents are either too close together or too far from each other, ensuring that the magnitude of the

penalties for clustering don't overshadow the contributions made towards containment, with an acceptable distance being mapped to a logarithmic curve to determine the rewards given:

$$\rho_{\text{agent}\to\text{agents}} = \begin{cases} -\max_f & \text{if } \|x_i - x_j\| = 0 \text{ or } \|x_i - x_j\| > r,\ i \neq j \\ \log(\|x_i - x_j\|) & \text{if } \epsilon < \|x_i - x_j\| \leq r,\ i \neq j, 0 < \epsilon < r \end{cases}. \tag{4}$$

in which ϵ is an arbitrary value to ensure our reward does not explode from the log function and $\max_f$ represents the maximum reward given to an agent. This same amount is taken away from their collected rewards if their Euclidean distances equate to zero or are greater than their optimal distance. If agents' distance to one another are anywhere in the range of the non-zero ϵ and optimal distance r, then their reward is determined by the logarithmic function which encourages spacing, and only slightly encourages distances that are at the bounds of what optimal spacing among agents was set to.

Additionally, a fixed positive reward $\max_p$, the maximum allowable reward between the agent's position x_i for the ith agent and the prey's position x_{prey}, is added to the agent's total reward for being within the desired distance r of the prey for $\rho_{\text{agent}\to\text{prey}}$, and the same amount is taken away to penalize the agent from straying too far from the prey. It was important to have the reward be fixed over the distance range from 0 to r so that the additional clustering reward could feasibly incentivize agents to cluster without being overshadowed by the rewards used to follow the prey. Various experimental trials have been done to ensure early termination as a result of system rewards didn't interfere with our results and that both rewards were scaled similarly so that weights of certain objectives reflect what we prioritize, containment first and structure second:

$$\rho_{\text{agent}\to\text{prey}} = \begin{cases} -\max_p & \text{if } \|x_i - x_{\text{prey}}\| = 0 \text{ or } \|x_i - x_{\text{prey}}\| > r \\ \max_p & \text{if } 0 < \|x_i - x_{\text{prey}}\| \leq r \end{cases}. \tag{5}$$

Concurrently, an acceleration cost implementation to our growing arsenal of reward functions has also been in development, showing good performance for following and capturing a prey with the Friendly Deployable Agents (agents) in the environment. The following function is a representation of our acceleration cost function:

$$\alpha_\tau = A_\tau \left(\frac{\tau\gamma}{t}\right), \tag{6}$$

$$R_\tau = R - \alpha_\tau. \tag{7}$$

in which α_τ represents the total acceleration cost at a given time step τ, the current acceleration action taken by the agent is represented by A_τ, the the total time steps in our epoch is represented by t, and γ represents the scaling factor for the time step interval for how we're enforcing this cost. The resulting reward R_τ at the current time step is then our cumulative reward R from Equation 3 decremented by the acceleration cost α_τ. The intuition behind this implementation is as the time step keeps increasing, so does the expenses for fuel or power, resulting in behavior that makes the agent more conservative with its fuel and forced to converge faster to offset the linearly scaling penalties for being in the environment. Another important component is also the magnitude of the acceleration action taken by the agent, which affects how quickly fuel is being consumed.

2. Future Work on the Reward Function

Reward function development has been at the heart of how the team influences agent behavior, whether the task is to follow the prey or maintain structural containment around the prey. The transition to acceleration-oriented actions from fully deterministic actions constrained to cardinal directions has helped simulate semi-realistic drone movement but has not been fully polished – actions appear to be immediate and jittery, disregarding the physical attributes of a drone in the field. More work is being done to better visualize our rewards dispersed and actions taken as part of our simulation, ensuring that every component of the simulation is behaving as intended and can help us understand why agents are behaving the way they are. So far creating a structure surrounding the agent has been the most complicated part of our reward function, pushing us to look into new metrics and decompose what we currently know works in addition to the progress being made to increase fidelity in the simulation in Section III.B.1. Some relevant literature that could impact how we proceed is Hindsight Experience Replay [14] (HER) as an alternative offline policy to our sub-optimal Proximal Policy Optimization (PPO) [15], and the use of ELO [16], a rating system method for calculating the relative skill levels of players in zero-sum games like chess named after Arpad Elo, or Trueskill [17] as per OpenAI's Hide and Seek's [18]

experimental findings for multi-agent systems. HER policy works with sparse rewards, which generalizes multiple goals by observing past actions made by the agent to infer useful information about the environment. This could useful as our agents have multiple goals to accomplish when training and changing policy implementation would allow us to view different behaviors that weren't possible in our previous policy. Identifying success solely by the system's rewards might also be a faulty and ambiguous metric, which is why rating systems such as ELO or Trueskill could be a better method to measuring performance improvements in agents through different policy versions and implementations. These approaches could also very well bypass the meticulous nature of shaping our reward function to our needs, for the time being we will continue to consider adding on different heuristics that can better route our agents to their goal state as shown by Figure 4 and improve both their physical and expected behavior.

B. Increasing Fidelity

The simulations in the previous section were run assuming basic behavior by the prey. In this section we augment that behavior in a number of ways. In order to provide the agents with a more realistic training environment, we modified the prey movement to evade the agents. Shown in Figure 6, an prey would move away from the agents by responding to the agents movements. If the prey entered a certain threshold (a threshold of 90 was picked keeping in mind that the

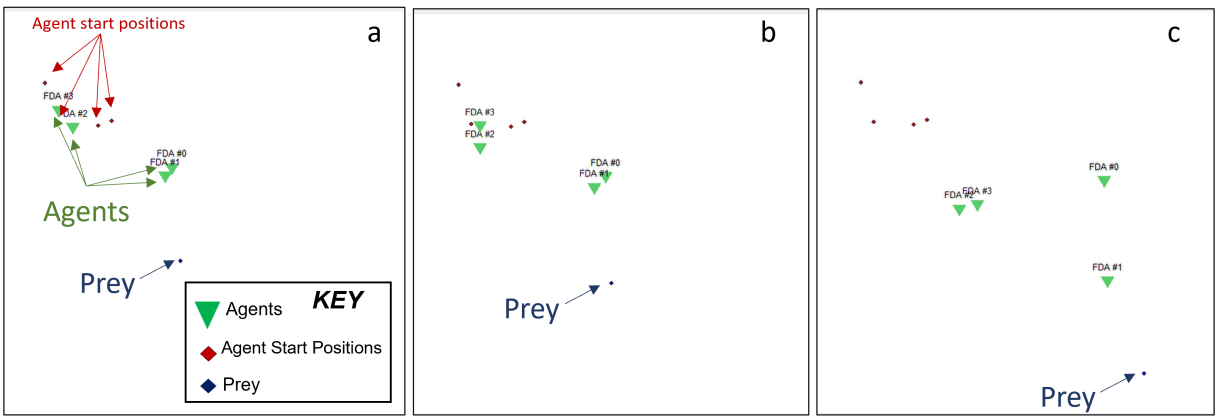


Fig. 6 (a) agents and prey at 2 seconds after agents move from start location. (b) agents and prey after 3 seconds. (c) agents and prey after 6 seconds.

"arena" size of the simulation is size 500) from the mean position of the agents, the prey would start to "evade". In order to make the prey evade, the algorithm would find both the mean position of the agents and the position of the closest agent, and determine if this position was within the threshold. If so, the preys velocity would be set to go in the opposite

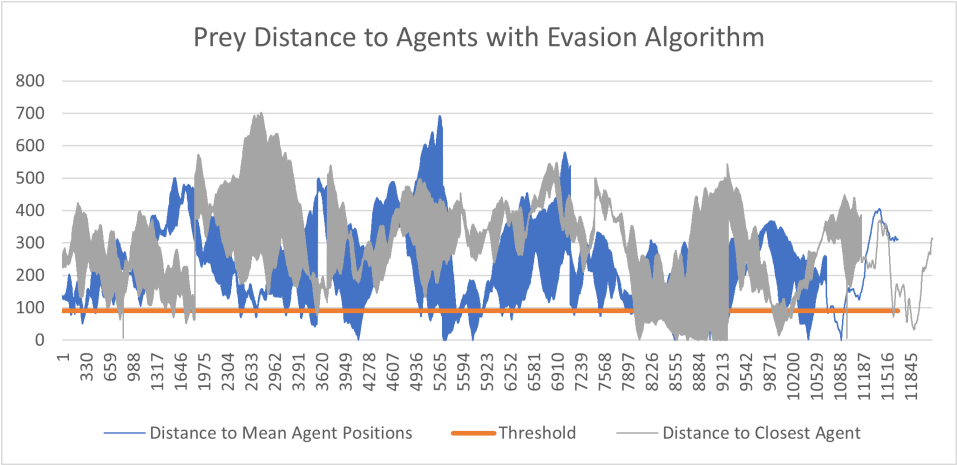


Fig. 7 Graph of prey distance to agents over time. Shows both metrics used (mean and closest agent).

direction (either left or right) of their position respective to the agent position. Figure 7 describes the movement of the prey within a single simulation, where the agents were trained to follow the prey. The prey's distance to the mean of the agent position or the distance to the closest agent must be above 90, shown as an orange line in Figure 7. As the figure shows, both distance parameters (distance to mean and distance to closest agent) rarely drop below the threshold.

We also modified our simulation environment to support multiple prey. There is now a parameter, similar to how we can define the number of agents, to choose how many prey are created in the simulation. Each prey still accounts for it's own `move()` and `step()` functionalities and has their own internal ID. Figure 8 shows a variety of prey added. The change was to create a more customizable training environment for the agents.

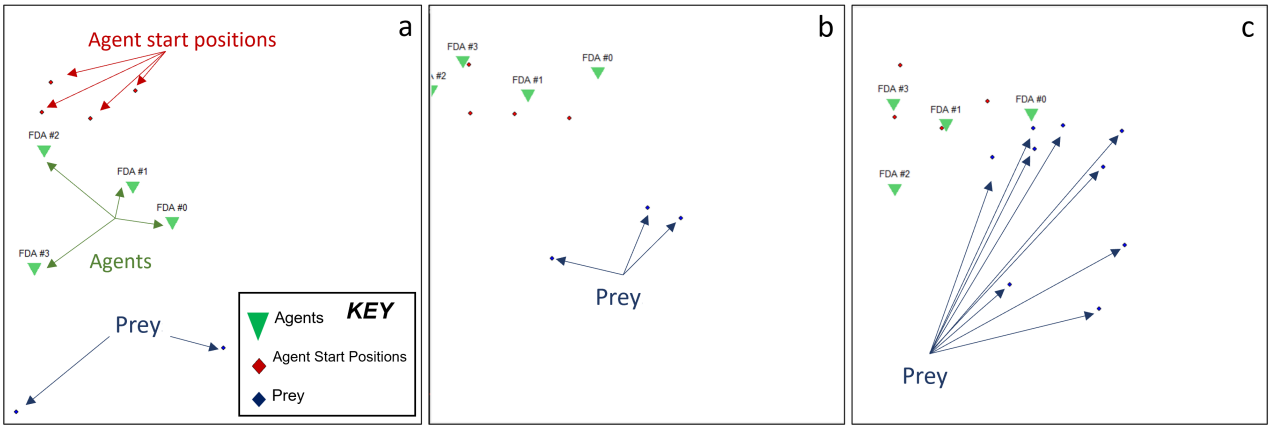


Fig. 8 (a) Simulation with 2 prey (shown with blue dots), (b) 3 prey, (c) 9 prey.

Our last goal in increasing fidelity was adding communication (Section III.C) between agents. This was implemented in successive steps, with each step a "layer" towards realistic communication between autonomous agents.

1. Future Work on Increasing Fidelity

The goal of increasing fidelity in the simulation is to be able to train the agents in more realistic situations. We have laid out a few next steps to continue to improve upon the prey behavior. The evasion algorithm currently in place is simple, and the prey often gets stuck in a corner of the simulation. In order to create a more continuous algorithm, the next step in the evasion strategy would be to set the prey velocity to the opposite of the mean velocity of the agents, as opposed to a position-based algorithm. With this change, when the agents have truly contained the prey and surrounded the prey on all sides, their average acceleration will be 0, and the prey will be unable to escape (Figure 9). This also leads to a new containment metric, where the prey has been contained once it is forced to have zero velocity and is unable to escape. RL methods have also been explored, where agents and prey can have competitive self-prey, with the prey being trained to evade the agents, and agents trained to contain the prey.

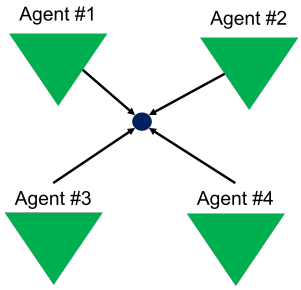


Fig. 9 Situation in which prey (blue dot) cannot move because net acceleration of agents is 0.

C. Communication

To facilitate increased fidelity of our multi-agent system simulation, we create a virtual transceiver and channel, atop which our simulated network infrastructure supports the operation of our distributed communication protocol.

1. Virtual Channel and Transceiver

We assume the virtual channel to contain the ideal amplitudes of a message's bits – that is, a bit 0 is modulated to amplitude -1, and a bit 1 is modulated to amplitude 1. A message's collection of modulated amplitudes are assumed to move together instantaneously as a one-dimensional block. We use a model of an omni-directional ideal transmitter, where the received signal strength (RSS) at any point within the communication radius is 100% of the transmission power, and 0% outside the communication radius.

The channel is multiplexed by the transceiver using Code-Division Multiple Access (CDMA), a multiple access method where every participating transmitter transmits, for every message bit, a series of pulses (code) multiplied by the bit's ideal amplitude (1 or -1). Should the chosen codes be orthogonal, a message sent into channel may be received / reconstructed losslessly. Each agent is allocated a row of a square Hadamard matrix as its "code". The dimensions of this matrix are $2^{\lceil \log_2(n) \rceil}$, where n is the number of agents in the simulation. Using this "code," the agent transmits its message for the given time step into the channel, avoiding collision entirely. As every agent knows the entire set of codes and their mapping to agents (where the ith agent contains the code corresponding to row i of the Hadamard matrix), each agent then attempts to parse every other agent's message as a packet. Should a given agent not be in-range, their "extracted transmission" will be all zeroes of a given length, corresponding to an invalid packet. This operation is embarrassingly parallel, as no communication or data dependency is needed between processes extracting different agents' messages (using their respectively orthogonal codes) from the channel simultaneously.

2. Network Infrastructure

Our packet contains the following fields. Bytes 00-02 and 02-04 are 2-byte source and 2-byte destination agent IDs, respectively. For broadcast communications, the destination address is set to 0xFFFF = 65535. Bytes 04-06 represent 2-byte flag field, where the most significant bit (MSB) indicates the message is fragmented, and the lower-order 15 bits are reserved for future use. Bytes 06-08 are a 2-byte field holding the communication round when the packet was first sent. Bytes 08-10 are a 2-byte field for the packet's "fragmentation sequence ID," such that fragmented blocks are ordered in sequence of their original, unfragmented message. Bytes 10-12 hold a 2-byte length of the payload, after the headers. Bytes 12-14 hold a 2-byte "hop count," the number of agents the packet has been forwarded through. Bytes 14 to the payload length hold the packets contents. The last four bytes of the packet after the payload represent the packet's 32-bit cyclic-redundancy-checksum (CRC32), to validate the packet has not been corrupted while transmitting.

An agent maintains two outgoing communication ring-buffers of capacity $numagents + 1$, sized to minimize queuing delay and packets dropped. These buffers are serviced using a round-robin weighted fair queuing method (WFQ) at a ratio of 2:1. Should a message be too large for a packet, it is fragmented into multiple packets, and all are inserted into the ring-buffer. If the fragmented message does not fit inside the remaining space of the buffer, none of the composite packet(s) are inserted, and the state of the buffer is left unchanged.

Upon successful reception of a packet by an agent, the packet is demultiplexed into the incoming buffer by communication round, source, and fragmentation ID. A agent's incoming buffer holds $numagents + 1$ "demultiplexers," which receive packet fragments, and emit a reassembled packet upon receipt of all fragments. Should a demultiplexer not receive all of its packet's fragments before falling off the end of the buffer, the packet is dropped. A non-fragmented packet is emitted immediately, skipping the incoming buffer entirely.

To enhance the security of the communication data, outgoing packets are encrypted using AES-256, a fast, symmetric block cipher, which are then decrypted on receipt by another agent. In this case, "symmetric" means that the same key is used to encrypt and decrypt the packet. We "preload" each agent with a copy of this key before the simulation begins, sidestepping the problem of key exchange over a potentially insecure channel. The monotonically-increasing "communication round" field of the packet makes a replay attack infeasible.

When a reassembled packet is "emitted" by the agent's incoming buffer, any number of callbacks may be called to process and/or route the packet further. For example, a naïve "gossip" protocol would route an incoming packet immediately onto the agent's outgoing bus, should space be available. In our protocol, the receipt of a packet triggers a state update callback.

3. Communication Protocol

The core of our distributed communication protocol is an "instance" – a 5-element vector consisting of an agent's instantaneous position $[x, y]$ and velocity $\left[v_x, v_y\right]$, alongside the time step the "instance" was recorded $[t]$. Naturally, an agent will broadcast its own "instance" as its current position and velocity, with the time step equal to the current simulation time step.

Each agent maintains a list of instances for every other agent in the simulation. Every time step, each agent broadcasts a message containing each of its maintained instances, along with its own (zero latency) instance out into the channel. The message's instances are ordered by ascending agent ID, as to remove the need for an ID field before each instance, saving space. Additionally, the prey's state is similarly maintained as an instance. If a prey agent falls within an agent's detection radius, it too is transmitted in the same manner.

Upon receipt of a message, the agent iterates through the message's instances. The agent compares the given instance's time step to its local instance for said other agent, and sets its record to the instance recorded closer to the present. As the maximum function is commutative, the processing order on simultaneous receipt of multiple instances for a given agent is irrelevant.

Upon stepping the Markovian decision process, each agent uses a first-order approximation (Equations (8) and (9)) based off of its instance to derive the fellow agent's current position and velocity.

$$V(instance, simulation) = instance.v_0 \tag{8}$$

$$X(instance, simulation) = instance.x_0 + instance.v_0 * (simulation.t - instance.t_0) \tag{9}$$

We define a "Communication Graph", Figure 10, to represent the state of our network, where agents are mapped to vertices, and an edge exists between two vertices if and only if the corresponding agents are within communication range of each other.

$$G = (V, E),\ V = v \in agents,\ E = (v0, v1) \in V,\ euclideanDist(v0, v1) <= r_{communication} \tag{10}$$

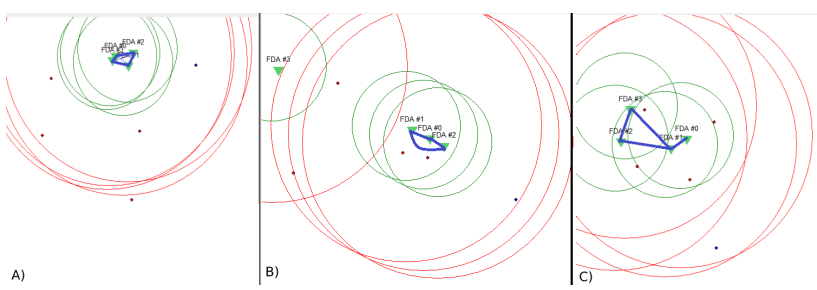


Fig. 10 (a) Network is fully connected, instance propagation takes 1 step. (b) Agent #2 is disconnected. (c) Network is connected, but instance propagation takes multiple steps. Blue lines are network links, green circles are the agents' communication radii, and red circles are the agents' prey detection radii.

We now show that, for a connected subgraph $g \subseteq G$ - that is, all the agents in the subgraph share at least one "edge," defined by being in communication range of the other - an agent's current instance will propagate through the entire subgraph in an optimum (minimal) amount of hops. As every instance is independently maintained in both the message and each agent's internal instance list, the propagation of instances themselves are independent from one another. Therefore, should one instance propagate optimally, *every* agent's instance information will propagate through the swarm in a minimum amount of time.

The propagation of an instance begins at its agent - this is the only source in the subgraph g which can "generate" a new instance (by reading its own position and velocity). At each time step, this instance can only ever propagate to an agent with a more out-of-date instance, as older information is auto-rejected by the algorithm's "max" call. This alone is sufficient to maintain loop freedom, as instances cannot flow "backwards" from the direction of propagation.

This above property is equivalent to the "visited" restriction on the breadth-first-search (BFS) algorithm – both ensure the search cannot visit the same vertex twice. As the broadcast operation is equivalent to "expanding" the front to all adjacent vertices in BFS, we are in essence performing a breadth-first graph traversal which is known to be efficient.

However, should multiple disconnected subgraphs exist, "new" instances from one subgraph's agents will *never* reach *any* agent in another - instead, the agent's record of all instances in the disconnected subgraph will grow slowly

out-of-date per time step (1 time step out-of-date per time step disconnected), a shortfall commonly referred to in network literature as the "count-to-infinity" problem [19]. However, in our current protocol and reliable channel, we may interpret an instance out-of-date by merely $numagents + 1$ time steps (assuming a worst-case graph diameter of $numagents$) to imply a disconnection, and allow our model to use this indicator to react in turn.

4. Future Communication Work

A more realistic Virtual Transceiver would include a continuous signal falloff function scaled with distance from the agent, as well as a random bit-error model to simulate environmental noise and other factors such as multipath fading. Additionally, the implementation of spread-spectrum modulation - using the existing CDMA scheme to spread the signal over multiple frequency bands - would significantly harden our networks against "jamming" attacks. [20]

The network currently operates using unreliable transport – callbacks to facilitate Automatic Repeat Request-based (ARQ) reliable transport are necessary for the guaranteed delivery of important messages. Additionally, the addition of error-correcting codes to *correct*, as well as detect bit-level errors are necessary upon the introduction of a "noisy" channel to minimize re-transmissions. Finally, local opportunistic forwarding - routing a packet to a subset of neighbors most likely to transmit successfully, based on a local estimation of global topology [21] - is necessary should unicast communication (agent to agent) be needed in addition to the current broadcast (agent to swarm) capability.

The future integration of high-fidelity sensors and cameras will necessitate a more complex protocol, capable of sending such readings in addition to instances. A better instance estimation algorithm is also possible, using leapfrog integration or a dedicated recurrent neural network (RNN). Finally, shaping the reward function using network QOS metrics (e.g. average packet delay, packet drop ratio, network disconnection) forms a powerful basis for future integration.

D. Porting to Unreal Engine

A primary motivation in the research being conducted is to create a simulation representative of real-world environments, physics, and other external factors. The current simulation visualization through a 2D Python graphical user interface (GUI) is an obstacle to achieving these goals without losing focus on the core of our research efforts. The solution to this issue is transferring the simulation training and visualization environment to a powerful 3D game engine with these high fidelity features already included - Unreal Engine.

To facilitate the transition from a pure Python simulation to an Unreal Engine frontend and Python backend simulation, several significant design changes were necessary. These design changes concerned the ability to control and access the world state of AirSim multirotor agents and other Unreal Engine objects in the Unreal Engine level (environment). AirSim is a Microsoft product that serves as an Unreal Engine plugin for simulating drones and other vehicles with integrated control physics, sensors, and more. AirSim control and access ability is possible through the Python AirSim application programming interface (API) which is integrated into the existing RL loop. For multirotor control, two alternatives were tested: control by velocity update and control by destination waypoint update. The former method involves translating the computed action to a velocity update and directly setting the multirotor's velocity vector in a 3D plane. The latter method involves translating the computed action to an updated position and setting a destination waypoint for the multirotor to automatically move towards the waypoint. These commands are delivered to the AirSim multirotors in parallel to ensure the commanding of one multirotor does not block or delay the commanding of another multirotor. These current control calculations could not be completed without real-time access of every active component in the Unreal Engine level.

The AirSim Python API enables access to the real-time state of the AirSim multirotors and any object in the Unreal Engine level at the time of compilation. These real-time states include vital information about collisions and estimated kinematics. Estimated kinematics include access to the multirotor's position, orientation, linear and angular velocity, and linear and angular acceleration. This information is used in calculations to determine the appropriate updates for the multirotor's velocity or destination waypoint as needed. For other Unreal Engine objects, the real-time state is referred to as the object's pose. The pose contains information about the orientation and position of the specified object. Another significant aspect of this implementation is the relative location between AirSim multirotor agents and Unreal Engine objects - the reference point to each multirotor or object location. The relative position can be set and standardized by means of specifying a "Player Start" actor anywhere in the level to set the absolute position of the origin.

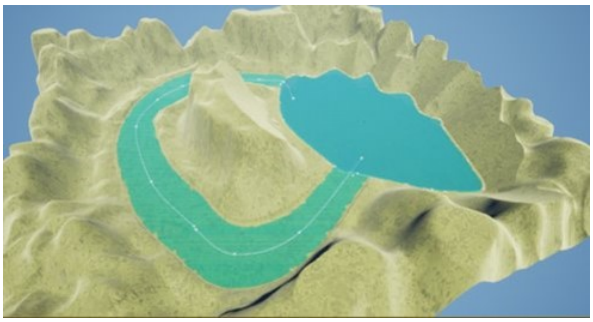
Following the establishment of an Unreal Engine frontend foundation for this simulation, several next steps are planned to further increase the overall fidelity of the system and prepare the simulation for critical next steps. Various control implementations for the prey will include direct pose updates, spline courses, manual control, and automated

movement paths. The RL observation and action spaces will drastically change. For the observation space, the 2D simulation is constrained to limited "sensors" for gathering knowledge about each agent's surroundings. The 3D simulation will create the opportunity for integrating numerous types of sensors to create heterogeneous agents and further expand the complexity and capability of the simulation – these capabilities are discussed in detail in the next paragraph. Similarly, the 2D action space lacks the physics-based fidelity – agents can turn a completely different direction with no acceleration transition or time expense (where expense represents a cost of performing a particular action). The 3D simulation will introduce the physics of drone movement through the AirSim Python API and will enable more realistic training through applied action expenses. The completion of this step in the development process of this project opens the door to multi-domain environment applications for testing 2D-trained RL algorithms (land, water, hybrid, etc.). More importantly, it serves as a transition point to a 3D training space to further advance the algorithm development and real-world application of this program's efforts.

E. Multi-domain Environments in Unreal Engine

Multi-domain training concerns the ability to generalize and train an RL policy in various unique Unreal Engine environments (levels) to optimize its behavior in unseen environments. While the RL policy shall be homogeneous and perform optimally across different environments, the computer vision weights must be substituted based on the current form of the prey. The significance of this effort is to support the concept of swarm deployment in unfamiliar territories while maintaining high performance and efficient task completion. Multi-domain training will involve the creation of various landscapes, such as desert, wetlands, mountain ranges, coasts, polar regions, valleys, oceans, and more. Each of these unique landscapes introduces a distinct challenge for the RL and computer vision algorithm in the form of physical obstacles and weather-related hindrances. Additionally, the implementation of these landscapes will involve the development of prey appropriately associated with the terrain.

Figure 11 represents our current work with developing additional levels for multi-domain training, beyond the original level developed presented in Figures 2 and 3. This level features water-characteristics such as a river and a lake, which will contain our prey and predators as water vehicles (such as a boats) as shown in Figure 11b. The river also flows counter-clockwise to the path shown in Figure 11a, which will test RL policy's environmental flexibility. As of now, we are tuning vehicle movement on water to match realistic water physics.



(a) Overhead view of our second custom Unreal Engine level, which features water-characteristics.

(b) Surface level of our Unreal Engine water-level, with an agent ship.

Fig. 11 An additional Unreal Engine level that is closer to our desired application space for maritime remote sensing.

F. High-Fidelity Sensors in Unreal Engine

To facilitate a 3D training infrastructure through Unreal Engine, both optical cameras as well as light detection and ranging (LIDAR) sensors will be utilized. These sensors are integrated into Microsoft AirSim multirotor agents. The optical camera provides three different camera views, pictured in (Figure 12), which can be leveraged for prey detection. To facilitate the automated prey detection, we plan to integrate the YOLOv5 object detection framework. YOLOv5 is the fifth official release of the You Only Look Once (YOLO) [22] object detection architecture. YOLO features real-time object detection by minimizing the number of neural network layers since each section of the input image is only

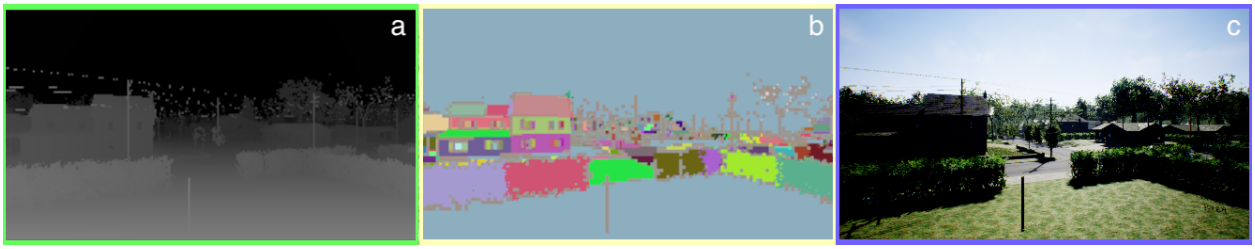


Fig. 12 Examples of camera views from AirSim multirotors. [6] (a) Depth view. (b) Segmentation view. (c) Optical view.

reviewed once, hence the name for the algorithm. YOLOv5 models will be trained and preloaded onto deployed agents for the corresponding environment domain and prey type. Additionally, these models will be trained on the camera's optical view to eliminate the delay of image processing required to convert the optical view to the depth or segmentation view. To train YOLOv5 models, our Unreal Engine team has used the created multi-domain environments to collect images of the various prey types from various perspectives and against distinct backgrounds to maximize the robustness of the model in new environments. The YOLOv5 model will be used in the remote sensing part of this project to facilitate detection of relative prey direction. Pairing this information with LIDAR sensor information, the information needed to construct the observation space of a particular agent for the 3D RL algorithm training infrastructure will be completed.

The LIDAR sensor available as part of the Microsoft AirSim plugin for Unreal Engine features a wide range of configuration options. The LIDAR sensor can have range, refresh rate, orientation, field of view (FOV), and many more settings precisely customized for the current use case. Additionally, a particular multirotor agent may have more than one LIDAR sensor mounted. The LIDAR sensor is implemented using ray casting, therefore, it returns a Point-Cloud as a flat array of floats along with the timestamp of the capture [6]. This Point-Cloud will enable an individual agent to determine the distance from itself to the prey when the prey is in the FOV of the sensor. Assuming the orientation of the multirotor optical camera and a LIDAR sensor are calibrated, the combination of the prey detection from the optical camera and the distance information from the LIDAR sensor can be utilized collaboratively in the swarm to derive an approximate location of the prey in the environment. This capability will enable our agents to effectively learn to achieve various advanced tasks including the identification, tracking, and containment of prey in any Unreal Engine level.

IV. Summary

Maritime remote sensing (MRS) in dynamic maritime environments requires adaptive solutions that are difficult to build in a deterministic context. Platforms tasked with monitoring maritime environments must be prepared to respond optimally to unfamiliar situations based on past experiences. Also, the usage of dispensable, simple, and decentralized agents is an attractive solution that minimizes cost, maximizes efficiency, and provides solutions to applications such as search and rescue, sanction avoidance, as well as protecting national security. Multi-agent systems (MAS) address each of these motivations when used in the context of a MRS swarm.

In this paper we began development of a reinforcement learning framework for multi-agent containment of one or more prey agents. We developed a two-dimensional environment in Python for prototyping and testing of reinforcement learning algorithms applied to the problem. We also created a three-dimensional environment based in Unreal Engine 4 with the Microsoft AirSim plugin to provide a more realistic environment for production demonstrations. For our reinforcement learning approach, we based our state space and action space on the Boids model of bird flocking behavior and developed a reward function to prioritize two agent behaviors: (1) tracking of the prey and (2) spacing between agents. We have further experimented with penalizing acceleration to model fuel consumption. To create a more realistic approach, we have begun adding (i) evasive prey movements (ii) more realistic communication protocols and (iii) multiple prey to the environment. For the three-dimensional environment, we have developed arenas based on a number of types of environments (desert, water, etc.) and demonstrated transfer of a trained policy from the 2D RL framework to have agents track the prey in three-dimensions. Future work is to incorporate the more realistic augmentations into the simulations, to find RL implementations that achieve more complicated and structured flocking behavior, and to incorporate computer vision so that training and tracking can occur with more realistic sensors.

Acknowledgments

This work is possible via a gift from the MITRE Corporation. The authors would also like the acknowledge the other team members that contributed to the technical implementation of this work: Antonio Alonso, Kenny Becerra, Austin Burcham, Mark Carroll, Aydin Gokce, Eashan Gupta, Sejal Gupta, Maryum Khan, Madison Martell, Aram Min, Andrew Neeser, Giang Nguyen, Vu Nguyen, Babatunde Ogundokun-Ayanda Jr, Kelechi Osueke, Audrey Ruckman, Shane Schipper Ashvita Vadicherla, and Haley Wisman. As well as technical mentors from MITRE Corporation: Mohammad Ridwaan Alam, Walker Lee Dimon, Chris Ward, and Mohammad Zarei. Lastly, we would like to acknowledge those that have helped keep the program running: Colleen Bartos and Ehren Hill.

References

[1] Alkhateeb, F., Al Maghayreh, E., and Aljawarneh, S., "A multi agent-based system for securing university campus: Design and architecture," *2010 International Conference on Intelligent Systems, Modelling and Simulation*, IEEE, 2010, pp. 75–79.

[2] Belkhala, S., Benhadou, S., Boukhdir, K., and Medromi, H., "Smart parking architecture based on multi agent system," *Int. J. Adv. Comput. Sci. Appl*, Vol. 10, 2019, pp. 378–382.

[3] Atınç, G. M., Stipanović, D. M., and Voulgaris, P. G., "A swarm-based approach to dynamic coverage control of multi-agent systems," *Automatica*, Vol. 112, 2020, p. 108637.

[4] Mora, T., Walczak, A. M., Del Castello, L., Ginelli, F., Melillo, S., Parisi, L., Viale, M., Cavagna, A., and Giardina, I., "Local equilibrium in bird flocks," *Nature physics*, Vol. 12, No. 12, 2016, pp. 1153–1157.

[5] Epic Games, `https://www.unrealengine.com`, 2019.

[6] Shah, S., Dey, D., Lovett, C., and Kapoor, A., "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," *Field and Service Robotics*, 2017. URL `https://arxiv.org/abs/1705.05065`.

[7] Chang, Y.-H., Ho, T., and Kaelbling, L., "All learning is local: Multi-agent learning in global reward games," *Advances in neural information processing systems*, Vol. 16, 2003.

[8] Le, N., Rathour, V. S., Yamazaki, K., Luu, K., and Savvides, M., "Deep reinforcement learning in computer vision: a comprehensive survey," *Artificial Intelligence Review*, 2021, pp. 1–87.

[9] Chamoso, P., Pérez, A., Rodríguez, S., Corchado, J. M., Sempere, M., Rizo, R., Aznar, F., and Pujol, M., "Modeling oil-spill detection with multirotor systems based on multi-agent systems," *17th International Conference on Information Fusion (FUSION)*, IEEE, 2014, pp. 1–8.

[10] Xu, F., Liu, J., Sun, M., Zeng, D., and Wang, X., "A hierarchical maritime target detection method for optical remote sensing imagery," *Remote Sensing*, Vol. 9, No. 3, 2017, p. 280.

[11] Qi, Q., Zhang, X., and Guo, X., "A deep reinforcement learning approach for the pursuit evasion game in the presence of obstacles," *2020 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, IEEE, 2020, pp. 68–73.

[12] Sapio, F., and Ratini, R., "Developing and Testing a New Reinforcement Learning Toolkit with Unreal Engine," *International Conference on Human-Computer Interaction*, Springer, 2022, pp. 317–334.

[13] Reynolds, C. W., "Flocks, Herds and Schools: A Distributed Behavioral Model," *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, Association for Computing Machinery, New York, NY, USA, 1987, pp. 25–34. https://doi.org/10.1145/37401.37406.

[14] Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O., and Zaremba, W., "Hindsight experience replay," *Advances in neural information processing systems*, Vol. 30, 2017.

[15] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O., "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[16] Elo, A., *The USCF Rating System: Its Development, Theory, and Applications*, United States Chess Federation, 1966. URL https://books.google.com/books?id=onUazQEACAAJ.

[17] Herbrich, R., Minka, T., and Graepel, T., "TrueSkill™: A Bayesian Skill Rating System," *Advances in Neural Information Processing Systems*, Vol. 19, edited by B. Schölkopf, J. Platt, and T. Hoffman, MIT Press, 2006. URL https://proceedings.neurips.cc/paper/2006/file/f44ee263952e65b3610b8ba51229d1f9-Paper.pdf.

[18] Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., and Mordatch, I., "Emergent tool use from multi-agent autocurricula," *arXiv preprint arXiv:1909.07528*, 2019.

[19] Schmid, A., and Steigner, C., "Avoiding Counting to Infinity in Distance Vector Routing," *Telecommunication Systems*, Vol. 19, 2002, pp. 497–514. https://doi.org/10.1023/A:1013858909535.

[20] El-Mahdy, A., Abou-Bakr, H., and Yasser, M., "Performance Analysis of DS-CDMA Receiver in Presence of Partial Band Jamming," *International Conference on Aerospace Sciences and Aviation Technology*, Vol. 13, 2009, pp. 1–9. https://doi.org/10.21608/asat.2009.23527.

[21] Biswas, S., and Morris, R., "ExOR," *ACM SIGCOMM Computer Communication Review*, Vol. 35, 2005, p. 133. https://doi.org/10.1145/1090191.1080108.

[22] Jiang, P., Ergu, D., Liu, F., Cai, Y., and Ma, B., "A Review of Yolo Algorithm Developments," *Procedia Computer Science*, Vol. 199, 2022, pp. 1066–1073. https://doi.org/https://doi.org/10.1016/j.procs.2022.01.135, URL https://www.sciencedirect.com/science/article/pii/S1877050922001363, the 8th International Conference on Information Technology and Quantitative Management (ITQM 2020 & 2021): Developing Global Digital Economy after COVID-19.