

3-24-2016

An OpenEagles Framework Extension for Hardware-in-the-Loop Swarm Simulation

Derek B. Worth

Follow this and additional works at: <https://scholar.afit.edu/etd>

Part of the [VLSI and Circuits, Embedded and Hardware Systems Commons](#)

Recommended Citation

Worth, Derek B., "An OpenEagles Framework Extension for Hardware-in-the-Loop Swarm Simulation" (2016). *Theses and Dissertations*. 326.
<https://scholar.afit.edu/etd/326>

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.



AN OPENEAAGLES FRAMEWORK
EXTENSION FOR
HARDWARE-IN-THE-LOOP
SWARM SIMULATION

THESIS

Derek B. Worth, Captain, USAF
AFIT-ENG-MS-16-M-052

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-16-M-052

AN OPENEAAGLES FRAMEWORK EXTENSION FOR
HARDWARE-IN-THE-LOOP SWARM SIMULATION

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

Derek B. Worth, B.S.
Captain, USAF

March 2016

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-16-M-052

AN OPENEAAGLES FRAMEWORK EXTENSION FOR
HARDWARE-IN-THE-LOOP SWARM SIMULATION

THESIS

Derek B. Worth, B.S.
Captain, USAF

Committee Membership:

Maj Brian G. Woolley, PhD
Chair

Dr. Douglas D. Hodson
Member

Dr. John M. Colombi
Member

Abstract

Unmanned Aerial Vehicle (UAV) swarm applications, algorithms, and control strategies have experienced steady growth and development over the past 15 years. Yet, to this day, most swarm development efforts have gone untested and thus unimplemented. Cost of aircraft systems, government imposed airspace restrictions, and the lack of adequate modeling and simulation tools are some of the major inhibitors to successful swarm implementation. This thesis examines how the OpenEagles simulation framework can be extended to bridge this gap. This research aims to utilize Hardware-in-the-Loop (HIL) simulation to provide developers a functional capability to develop and test the behaviors of scalable and modular swarms of autonomous UAVs in simulation with high confidence that these behaviors will propagate to real/live flight tests. Demonstrations show the framework enhances and simplifies swarm development through encapsulation, possesses high modularity, provides realistic aircraft modeling, and is capable of simultaneously accommodating four hardware-piloted swarming UAVs during HIL simulation or 64 swarming UAVs during pure simulation.

*To my mom, your love and support throughout my life has positively impacted me
and is without a doubt why I am where I am today.*

*To my wife, I can't express into words how grateful I am for your enduring support
and encouragement during the past eighteen months. Thank you for patiently and
lovingly lifting me up from my daily gloom.*

Acknowledgements

I would like to thank my advisor, Maj Brian Woolley, for his visionary guidance during the thesis process. Your flexibility and clear feedback made you invaluable throughout the past year. I would also like to thank Dr. Douglas Hodson for dedicating his time both in and outside the classroom. Your insights and one-on-one instruction were indispensable as I learned and applied the OpenEagles framework.

Derek B. Worth

Table of Contents

	Page
Abstract	iv
Dedication	v
Acknowledgements	vi
List of Figures	ix
List of Tables	xi
I. Introduction	1
1.1 Research Objective	2
1.2 Overview	3
II. Background	4
2.1 Why UAV Swarms?	4
2.2 The Need for a Simulation Framework	6
2.3 Current State of UAV Swarm Simulation	8
2.3.1 Study of Swarms	8
2.3.2 Essential Simulation Components	12
2.3.3 Hobbyist Community	14
2.3.4 Simulation-Chaining	19
2.4 Quality Attributes	21
2.5 OpenEagles	23
2.5.1 Model View Controller Design Pattern	23
2.5.2 Base Classes	26
2.5.3 Eagles Definition Language	27
2.5.4 Main Methods	29
2.5.5 Extending Built-in Functionality	30
2.6 Summary	32
III. Framework Design and Analysis	33
3.1 Swarm Algorithm Placement	33
3.2 Communication Flow	36
3.3 Extending OpenEagles	37
3.3.1 Enabling Swarm Behaviors	38
3.3.2 Enabling Hardware-in-the-Loop	39
3.3.3 The Swarm Simulation Framework	41
3.4 Framework Assessments	43
3.4.1 Assumptions and Limitations	43

	Page
3.4.2 Qualitative Measures	45
3.4.3 Quantitative Measures	45
3.4.4 Swarm Development Scenario	48
3.5 Summary	52
IV. Design Demonstrations/Results	53
4.1 Applying Reynolds Flocking Rules	53
4.1.1 Setup	53
4.1.2 Observations	55
4.2 UAV Swarms with Simulated Autopilots	58
4.2.1 Setup	58
4.2.2 Observations	59
4.3 Applying Hardware-in-the-Loop Simulation	62
4.3.1 Setup	62
4.3.2 Observations	65
4.4 UAV Swarm with Hardware Autopilots	69
4.4.1 Setup	69
4.4.2 Observations	70
4.5 Summary	71
V. Conclusions and Future Work	72
5.1 Final Remarks	74
Bibliography	75

List of Figures

Figure	Page
1	Swarm Control Models 8
2	Example UAV Hardware Architecture 13
3	Diverse Airframes 16
4	Hardware-in-the-Loop 18
5	Simulation-Chaining 20
6	Simulation-Chaining using FlightGear 21
7	OpenEagles Design Pattern 24
8	UAV System Hierarchy 25
9	OpenEagles Base Classes 27
10	Example EPP File 28
11	Eagles Definition Language Parsing 30
12	OpenEagles Package Hierarchy 31
13	UAV Mission Overview 34
14	Communication Diagram of Swarm Simulation Components 37
15	UAV Systems Communication Diagram 39
16	UAV Systems Communication Diagram (with HIL) 40
17	Swarm Framework Class Diagram 42
18	Simulation Run-time Graph 46
19	Sample Performance Graph 47
20	Scenario Common Waypoint Field 50
21	UAV Paths 50
22	Sig Rascal 110 ARF Model 51

Figure		Page
23	SIMDIS Overview	52
24	Reynolds Flocking Rules	55
25	Flight Maneuvers	56
26	Simulated Reynolds Vectors	57
27	Performance Graph - Single Swarming UAV	58
28	Swarm Size Performance Comparison	60
29	Real-time Swarms	61
30	Large Swarm	61
31	Pixhawk Hardware Setup	63
32	Dynamic Waypoint Following with QGroundControl	64
33	Performance Graph - Single Swarming UAV (HIL)	67
34	Reynolds Vectors with HIL Simulation	68
35	Multi-Pixhawk Hardware Setup	69
36	Swarm Size Performance Comparison (HIL)	70
37	Swarming During HIL Simulation	71

List of Tables

Table		Page
1	Swarm Studies	10
2	Hobbyist Community Tools	15
3	Percent Real-time Execution	59
4	HIL MAVLink Message Traffic	65
5	Percent Real-time Execution (HIL).....	70

AN OPENEAAGLES FRAMEWORK EXTENSION FOR HARDWARE-IN-THE-LOOP SWARM SIMULATION

I. Introduction

Unmanned Aerial Vehicle (UAV) swarms show promising potential in various applications such as surveillance and reconnaissance, counter swarm defense, search and rescue, surrogate communication relay, and rapid geography mapping [43, 48]. Although the use of swarms for such applications is possible in theory, actual implementation and successful deployment proves costly and highly challenging. In recent years, modeling and simulation has proven itself to be an important initial phase in the development process that mitigates cost and sets the foundation for successful deployment. The need for a simulation environment to test and develop UAV swarms is paramount to their realization in real-world applications.

Many multi-agent simulation frameworks already exist that could potentially, after some modification, meet the requirements necessary to model UAV swarm behaviors. However, many of these require significant source code development and overhaul in order to simulate only simple aspects of a swarm or a specific swarm scenario. It is desirable that a swarm simulation framework be complete enough to simulate simple aspects of a UAV swarm—such as vehicle types, positions, and control inputs—while providing developers a workspace or “playground” to insert their sensor packages and swarming algorithms. A framework with built-in reusable and applicable functionality reduces overhead and upfront workload, allowing developers to concentrate on *swarm* development instead of *simulation* development.

Unfortunately, the diversity and complexity of UAV swarms will not allow swarm developers to circumvent simulation development altogether. It is impossible to create a general purpose or “one-size-fits-all” solution that accounts for every potential aspect of a swarm. Instead, more benefit is gleaned from identifying, categorizing, and incorporating common patterns, configurations, and constructs as well as widely-accepted processes, protocols, structures, and architectures that have already proven successful. Additionally, the framework must implement useful code that will remain unchanged by users (the swarm developers), while providing extensibility (i.e. can be extended/overridden). This thesis explores ways to establish such a framework for UAV swarm development.

In addition, a technique known as hardware-in-the-loop (HIL) enhances simulations by incorporating hardware embedded systems that control actual UAVs. Specifically, the simulation provides sensor information (e.g. position, attitude, velocity) to the hardware autopilot in exchange for control information (i.e. stick, rudder, throttle) from the autopilot back to the simulation. HIL simulation alleviates the need to make assumptions about autopilot performance and forces the simulation to account for timing and bandwidth constraints associated with communication protocols, firmware implementations, processor speed limitations, and onboard applications integrated into the autopilot. Incorporating the actual hardware device used aboard real flights increases confidence that swarms in live flight testing will behave as modeled in simulation. Thus the swarm simulation framework explored in this thesis is constructed to accommodate HIL.

1.1 Research Objective

Ultimately, the objective of this work is to demonstrate a simulation framework utilizing HIL that allows developers to develop and test the behaviors of scalable and

modular swarms of autonomous UAVs in simulation with high confidence that these behaviors will propagate to real/live flight tests. This research effort seeks not to invent a new framework, but instead extend an existing one that will accommodate swarm behaviors as described above. Specifically, the intent of this effort is not to develop swarm algorithms, but define a space where swarm developers can develop and test their own. This thesis leverages existing technologies and research advancements in the fields of simulation, swarm control theory, visualization (computer graphics), flight dynamics, serial and link layer communications, and HIL integration.

1.2 Overview

The remainder of this thesis is outlined as follows: Chapter II provides context for this research effort to include justification for pursuing a swarm simulation framework as well as elicitation of its requirements, relevant design considerations, and an introduction to the OpenEagles simulation framework. Chapter III outlines how OpenEagles is extended to accommodate swarm behavior development and HIL simulation. The proposed swarm simulation framework is put to the test in Chapter IV where a swarm development scenario is used to demonstrate its capabilities and limitations. This chapter reveals that high-rate data translations between simulation and hardware devices induce a bottleneck to swarm scalability during HIL simulation. Lastly, Chapter V closes with a final assessment of the swarm simulation framework and recommendations for future work.

II. Background

The design and implementation of any software framework demands a deep understanding of requirements the software must satisfy. This chapter elicits those requirements by first examining why a swarm simulation framework is necessary and what past swarm development efforts have taught us about the common elements of a swarm. Lastly, current technologies and design considerations relevant to the implementation of a swarm simulation framework—many of which serve as foundational components of the proposed framework—are discussed followed by the introduction of the OpenEagles simulation framework.

2.1 Why UAV Swarms?

If you are reading this paper, you may be asking yourself, “Why are UAV swarms useful or important?” Unfortunately, the current widespread use of UAV swarms only exists in the imagination of visionary thinkers of the future, and thus no true measure of utility has been established. With the development of microcomputers, artificial intelligence, wireless network communications, and the exploration of autonomous flight, only recently has technology matured enough to make such visions a reality. This section attempts to address why swarms are important by exploring opinions and advice from experts in relevant fields such as disaster relief and military strategy. But first, what are UAV swarms? As described by the PennWell Aerospace and Defense Media Group, “UAV swarms are groups of UAVs that work together to accomplish goals, communicating with each other and assisting other members of the swarm in tasks” [15]. The general consensus in UAV swarm utility is the ability to do more with less. That is, require fewer operators to accomplish large and complex missions cheaper, faster and more efficiently with fewer mistakes.

Kumar [23] cites search and rescue as one promising area that swarms would obviously have an advantage over single aerial vehicles. A swarm could autonomously coordinate and cover more ground faster and with only a single operator. Imagine dozens of UAVs, combing an area, relaying information not only to the operator, but amongst each other, coordinating the prevention of duplicate coverage. Swarms configured to perform search and rescue missions have a greater probability of finding victims than a single aircraft. Additionally, the autonomous coordination of the UAVs frees the operator to make higher level strategic decisions such as diverting the swarm (or parts of the swarm) to a higher priority area based on new external intelligence received. This should benefit recovery efforts after natural disasters, ultimately expediting rescue response time and treatment of victims.

In military applications, swarms could play an integral role in both offensive and defensive measures during warfare. Scharre [43] argues that large numbers of UAVs have several potential advantages in combat, including combat dispersion, swarm resiliency, graceful degradation, and enemy defense saturation. These advantages are evident when considering swarm attacks on current air defense systems, such as surface-to-air missile (SAM) sites, which are typically postured to engage only a few sequential targets at a time. Current SAM sites are not designed to defend against overwhelming numbers of air vehicles attacking in unison and often rely on missiles that are equipped with expensive payloads, guidance systems and sensors—swarms prove economic dominance by drawing out million dollar missiles from the enemy arsenal for the price of UAVs worth a few thousand dollars. Such factors grant swarms the advantage in combat. Armed swarming UAVs will be inexpensive and disposable, providing an economical and more reliable means of defeating enemy defense systems. Chung [23] suggests that the only army big enough to stop a swarm is another swarm.

Besides saving lives during search and rescue operations and destroying enemy defenses on the battlefield, UAV swarms show promising opportunities in other areas to enable coordinated actions and effects previously thought impossible. We will only realize and expand upon the true value of those opportunities once swarming behaviors have been developed, tested, and put into practice. The next section explains the need for simulation as a bridge between the idea of swarms and their implementation in the real world.

2.2 The Need for a Simulation Framework

Swarm configurations vary widely—from simple and orderly (such as a grid formation) to highly dynamic and seemingly chaotic (such as a flock of UAVs forming a tight ball and traveling as one while avoiding collisions). To develop and test such a wide range of swarm configurations, modeling and simulation is necessary. Bypassing the modeling and simulation phase of the development process is costly. Live UAV flight testing comes with many associated resource requirements. Some of these include the inherent cost of the physical components such as the ground station, vehicle, fuel, and sensor packages.

Additionally, extrinsic factors such as restricted airspace limit live flight testing. The U.S. has the busiest, most complex airspace in the world [1] and developing and testing real UAV swarms in this congested domain presents challenges for both the FAA and aviation community—the cost of which is made evident in the following example: An 18-month trial testing UAVs for agricultural utility (which concluded in June 2015) required a variety of small UAVs with price tags ranging anywhere from \$9K to \$100K depending on the sensor packages needed as well as an additional average operating cost of \$30 to \$50 per hour. Furthermore, the costly and lengthy government approval process ultimately resulted in tight restrictions where each UAV

had to remain within line of sight, always below 400 feet AGL, and weigh less than 2 kilograms [14]. This example demonstrates some of the challenges of real UAV flight testing that are easily mitigated in simulation. Simulating the air vehicles and sensor packages not only greatly reduces monetary costs, but also opens the exploration of tests outside the limitations of government imposed restrictions.

Alternatively, system developers turn to modeling and simulation. While some existing simulators do support UAV swarms to a limited extent, most rely on hefty assumptions that do not translate well to real flight. Corner [11] provides a survey of many relevant swarm and network simulators and simulation frameworks. While all provide isolated modeling of specific UAV swarm aspects such as animated particle swarming, UAV/target interactions, and statistical simulation of ad-hoc networking between swarm nodes, none provides an accurate and comprehensive flight simulation environment ready to model a variety of airframes configured with customized autonomous behaviors. As a result, swarm development continues to flourish only in theory.

Developers have already proposed swarm-centric solutions for performing missions such as multi-target detection and localization [3, 12, 33], multi-target tracking [27], persistent sensing [40], and reconnaissance/surveillance [42]. Some have even shown how UAV swarms can enable tracking of environmental conditions such as the movement of harmful ocean debris [41] and contaminant cloud boundaries [44]. However, such swarm behaviors thus far remain unimplemented. A simulation framework capable of robust and accurate modeling of UAV swarms is necessary to overcome the above assumptions and challenges. By providing a means to accurately simulate UAV swarms, swarm developers can initialize, run, and rerun a variety of tests at a fraction of the time and cost of real flight testing without its limitations. Only then can developers prove or disprove the viability of swarm-centric solutions.

2.3 Current State of UAV Swarm Simulation

The use of UAV swarms in real-world applications is a relatively new concept. Only in the past few decades have we toyed with the idea and began dabbling with its implementation. During that time, many pioneers have tested theory by advancing control strategies, algorithms, and simulations necessary to make UAV swarms a reality. Here, we will examine many of those efforts, some of which serve as a foundation for the proposed swarm simulation framework.

2.3.1 Study of Swarms.

People have studied different aspects of swarms, trying to answer questions like “How can we control swarms?” and “Of the different swarm control strategies, which one is the best?” Of course the answer to the latter is, *It depends!* As seen in Figure 1, there are a variety of control strategies that could prove more or less suited for dif-

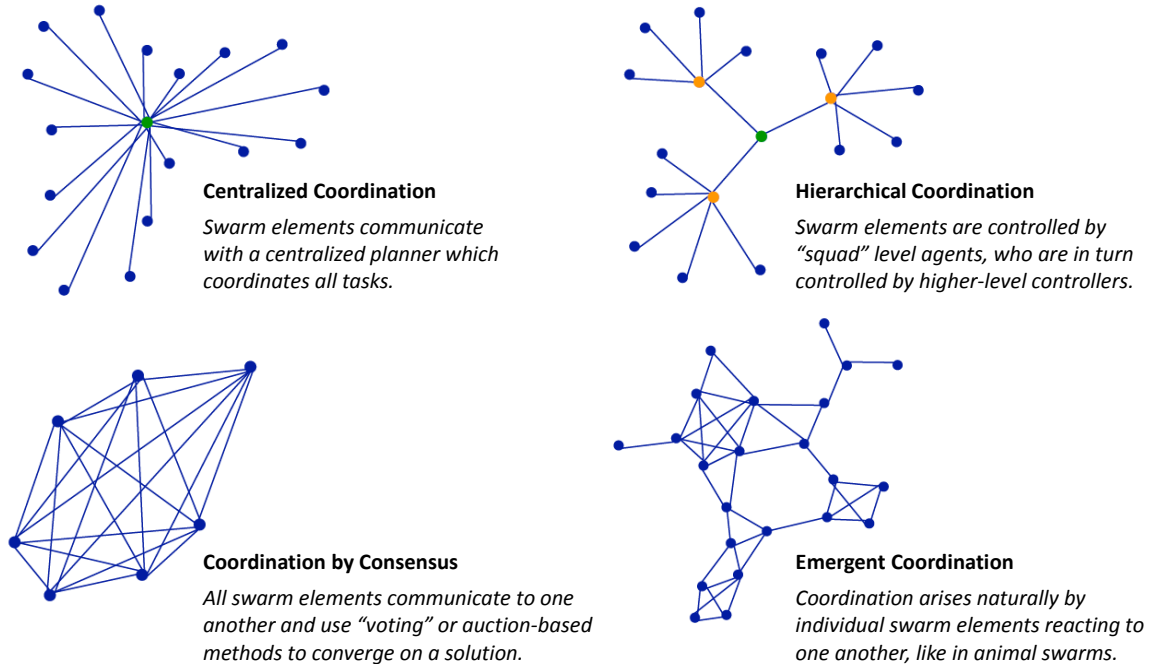


Figure 1. Swarm Control Models. (Originally presented in [43] and reproduced as an element of the public domain.)

ferent scenarios. A swarm using emergent coordination is more resilient during an attack mission—achieving graceful degradation when taking on enemy fire—than one, for example, using centralized coordination where the enemy only needs to destroy the lead UAV to take out the entire swarm. But how do we know if implementing and applying swarms with emergent coordination to accomplish offensive actions is feasible? And if it is, will it prove more beneficial than using other means?

Many swarm developers model their algorithms and control strategies in simulation to answer these questions. Some use robust high fidelity simulation environments while others use simple mathematical models and visualize results using software like the popular MathWorks[®] tool, Matlab. Table 1 shows a list of various swarm studies that have taken place over the past 15 years. As you can see, a wide variety of approaches were used to model the algorithms or swarm control strategies studied. When comparing modeling approaches of these studies, a few notable patterns emerge. First, simulation complexity is inversely related to the quantity of UAVs simulated. Simulations with higher complexity modeled fewer UAVs (~ 3) while simple simulations were capable of modeling many more ($+30$). This makes intuitive sense since high fidelity modeling requires more processor resources to compute the finer detailed characteristics of aerodynamics, sensing, and environmental effects. It is also worth noting that studies using complex modeling relied on foundational tools such as highly matured flight simulators and simulation frameworks rather than custom tools built for a specific purpose. By using accurate tools that are widely used and in constant development by the community, swarm developers gain confidence and trust that those tools are adequately validating or disproving their hypotheses.

This does not mean all swarm development efforts require such accuracy. Although many of the studies listed in Table 1 validate hypotheses using only simple models in conjunction with many assumptions, the results from those studies do provide a

Table 1. Swarm Studies

<i>Swarm Study</i>	<i>Year of Study</i>	<i>2D/3D Simulation</i>	<i>Max UAV Count</i>	<i>FDM/Airframe</i>	<i>Communications Implemented</i>	<i>Sensors Implemented</i>	<i>HIL Implemented</i>	<i>GCS Implemented</i>	<i>Simulator Custom Built</i>	<i>Foundational Tool(s)</i>	<i>Visual Interface</i>
Cooperative Control [37]	2002	Both	8	C	Y	Y	N	N	N	Simulink/ MultiUAV	Matlab Plots and AVDS playback
Wide Area Search [8]	2002	2D	8	S	N	Y	N	N	Y	Matlab and Simulink	Matlab
SEAD and Target ID [20, 21]	2003/5	3D	32	S	Y	Y	N	N	Y	Custom	Command Line Feedback
Target Locating and Tracking [25]	2005	2D	U	VS	N	N	N	N	Y	Custom, Java programming	Rectangular grid with GUI controls
General Flocking Algorithms [34]	2006	Both	150	U	U	U	N	N	Y	Unspecified	2D/3D Graphical Plots
Vehicle Routing Problem [30]	2007	Both	640	S	N	Y	N	N	N	SPEEDUS	Map Overlayed with Data
Search and Destroy [32]	2007	2D	65	S	Y	Y	N	N	N	SWARMFARE	Script Visualization
Formation Flight [19]	2010	3D	3	VC	Y	N	N	N	N	X-Plane/Matlab/ Simulink	3D GUI with Matlab Scripting
Flocking and Comm Range [24]	2011	3D	10	S	Y	N	N	N	Y	Unspecified	2D Graphical Plots
Search and Attack [9]	2012	2D	50	VS	Y	Y	N	N	N	SWARM Simulation	2D Map GUI
Swarm-vs-Swarm [17]	2013	3D	100	C	Y	Y	N	N	N	MASON	3D GUI
Point ISR Flocking [28]	2014	3D	10	S	N	Y	N	N	Y	Matlab	Matlab Plots
Formation Flight [49]	2014	3D	4	VC	Y	N	N	N	N	FlightGear/ Matlab/Simulink	Matlab Plots and Flight Sim GUI
Flocking for Drag Reduction [29]	2014	2D	15	VS	N	N	N	N	Y	Matlab	Matlab Plots

VS (Very Simple) - modeling using only discrete/incremental 2D positioning; models typically move in a linear fashion

S (Simple) - modeling using discrete/incremental 2D or 3D positioning

C (Complex) - modeling using mathematical equations for continuous 3D positioning, accounting for the six degrees of freedom

VC (Very Complex) - using fully functional flight dynamics models, accounting for aerodynamics and environmental effects such as gravity and weather conditions

U (Unspecified) - no functionality/attribute specified

wealth of knowledge. However, when moving from simulation to real-world flight testing, comprehensive and accurate simulations are necessary. For example, Hexmoor [25] explores how a bidding process can enable coordinated multi-target tracking. The developers in this study follow a detailed experiment to test theory in simulation. However, the simulated environment consists of small shapes of different colors and sizes (representing the UAVs and targets) that move with only two degrees of freedom (i.e. vertically and horizontally) across a small rectangular grid. The study does produce some meaningful conclusions about the bidding process, but is hardly a test of how the process will translate to a real swarm of UAVs.

Another notable observation is that of all the studies, half relied on custom built simulators and only two did not use an existing framework or tool set as a foundation for the modeling environment. Also notice that almost half of the simulations relied exclusively on Matlab/Simulink for data generation and/or interpretation. This indicates the need for common simulation tools that are accurate and easy to use. Swarm developers are less likely to explore certain swarm behaviors if there are no tools available to test their theories or the tools are too complicated to use. Also, custom-built tools often rely on simple designs and too many assumptions to reach valid conclusions about swarm control algorithms/strategies under observation. A swarm simulation framework should serve as a common tool that is easy enough to extend and implement custom scenarios, but accurate enough to provide useful results.

Lastly, note that none of the studies implemented hardware-in-the-loop (HIL) or swarm ground control station (GCS) functionality, which reflects the current state of UAV swarm research. That is, current swarm control algorithms and strategy are not yet mature enough to warrant testing of such advanced functional swarm components.

Nonetheless, as discussed in later sections, implementing these are largely beneficial when transitioning from theory to actual UAV swarm flight testing.

2.3.2 Essential Simulation Components.

Studies in Table 1 represent the current approaches to simulating UAV swarms. Each study addresses a specific aspect of swarm behavior development, but does not provide a comprehensive solution capable of translating directly to real-world behaviors. The objective of this thesis is to propose a framework design that accommodates HIL simulation in addition to all components necessary for accurately modeling UAV swarms—thus providing a comprehensive solution. This section outlines three essential simulation components (entities, subsystems, and views) that are common to all swarm simulations while the next section introduces technologies developed in the hobbyist community that make up the remaining components necessary to enable accurate HIL simulation of swarms.

Entities. The first simulation components evident among all swarm simulations are software defined *entities*. Many of the swarm studies in Table 1 simulate not only UAVs, but also external players and targets. For example, Gaerther’s study [17] requires a distinction between allied and enemy UAVs to show how two separate swarms will interact with each other. In this case, two different entity types (i.e. allied and enemy UAVs) are required. Additionally, many of the scenarios require target entities. Software defined entities enable the modeling of various “players” that are critical to swarm development in simulation.

Systems and Subsystems. All scenarios modeled in Table 1 indicate that swarming UAV entities (potentially non-UAV entities as well) must interact with their environment and each other. Modeling corresponding sensory and communication functions in simulation necessitates the incorporation of entity *systems and*

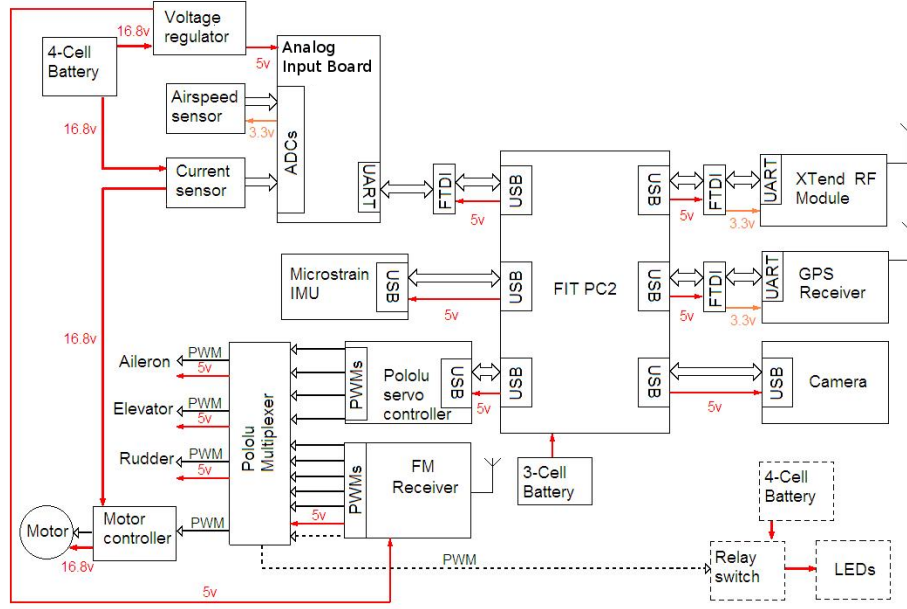


Figure 2. Example UAV Hardware Architecture. A UAV is comprised of many subsystems; in simulation, developers should have the ability to model these as needed. (Originally presented in [47] and reproduced as an element of the public domain.)

subsystems. The example UAV hardware architecture depicted in Figure 2 supports this claim by showing real UAV systems and subsystems, such as onboard sensors and communication interfaces that collect and exchange information with operators and other UAVs. Sensors, such as Global Positioning System (GPS) receivers, cameras, and inertial measurement units (IMUs) capture external input (e.g. position, speed, imagery, attitude, etc.) necessary for the autonomous decision making that is inherent with swarm behaviors. This sensory information is then processed and transmitted not only over communication links to operators on the ground and other UAVs, but also to other components onboard the UAV. Since information gathering and dissemination is the primary objective of most UAV missions [13], these systems and subsystems are vital to both real and simulated UAV operations.

Views. Another important simulation component evident in all swarm scenarios is a *view* or visual interface to the simulation. Simulation data is difficult to interpret

without a means to visualize and interact with it. For example, it is much easier and intuitive to detect when an aircraft is performing a specific maneuver (e.g. nose-high stall or barrel roll) by observing its motion in 3D space rather than analyzing a table of time, position, velocity, and attitude measurements. At a minimum, a visual interface to the simulation must be capable of receiving simulation data and translating it to a meaningful visual representation of the swarm. Some useful aspects of visually representing a swarm are rendering of the UAVs in 3D space as well as recording and speed-controlled playback. 3D rendering should include the ability to pan, tilt, zoom, and rotate views around the swarm or individual UAVs in the swarm. Some visualization frameworks provide “fly by” and “follow” views which enhance simulation analysis. Other information that the view should readily visualize are individual aircraft state parameters (e.g. altitude, speed, roll/pitch/yaw angles), flight paths (i.e. contrails behind UAVs showing position at previous times), and UAV relationships (e.g. separation and altitude differences between UAVs). To address these requirements, the swarm simulation framework should accommodate many different views.

2.3.3 Hobbyist Community.

Many drone hobbyists find UAVs fun and exciting to “play with” in their spare time. They share what they have tried on blogs and other public forums, helping each other solve drone related problems. DIY Drones (<http://diydrones.com/>) is a prime example of such public forum in which amateurs from all over the world come together to share their custom UAV designs. This open source community provides a wealth of knowledge and experience derived from a vast pool of collective efforts, testing new ideas through trial and error. Through such efforts, many useful tools and techniques have emerged that are relevant to swarm simulation.

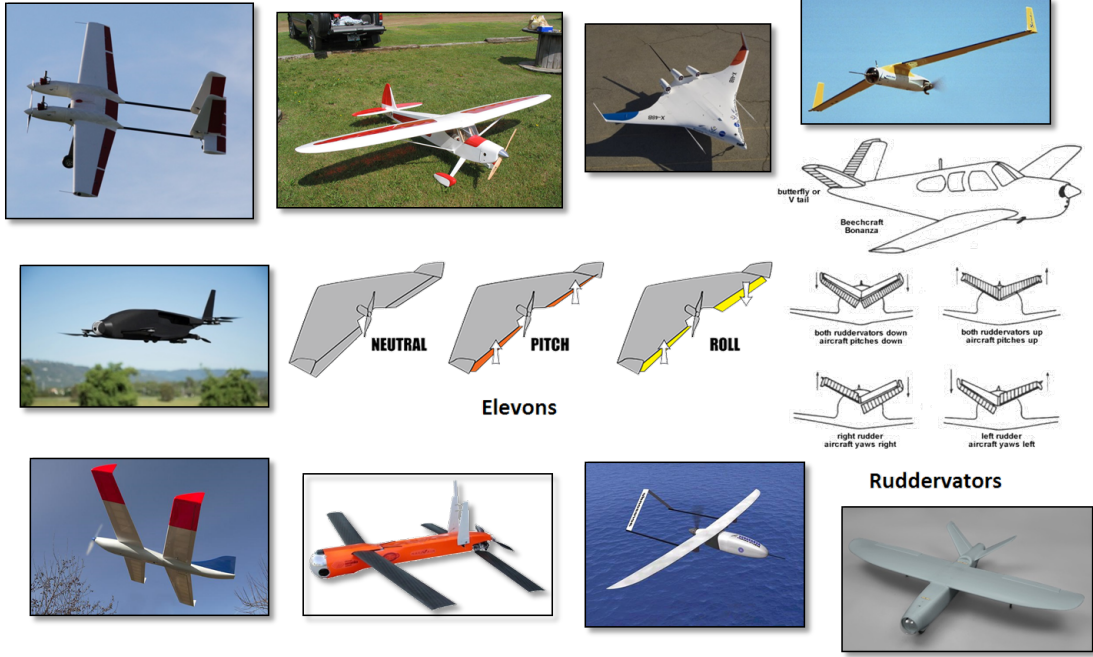
The community has produced remote control (RC) aircraft, GCS software, autopilot hardware/firmware, high-throughput/low-overhead communication protocols, and flight simulators that, when used together, provide a variety of capabilities. Table 2 lists some of the tools available in the hobbyist community today.

Table 2. Hobbyist Community Tools

Type	Tools
<i>RC Aircraft with FDM</i>	HiLStar17F/EasyStar, Zagi flying wing, Sig Rascal 110, Viper X-10, Multiplex TwinJet, Early Bird glider
<i>Ground Control Station</i>	MAVProxy, QGroundControl, Mission Planner, APM Planner
<i>Autopilot</i>	OpenPilot, FlexiPilot, ArduPilotMega, SLUGS, Pixhawk/PX4, MatrixPilot/UAVDevBoard, SMACCPilot, Armazila, Aerob
<i>Communication Protocol</i>	MAVLink, UAVCAN
<i>Flight Simulator</i>	XPlane, FlightGear, Microsoft Flight Simulator, GEFS Online, AeroSimRC, JSBSim

Airframe. Table 2 lists a small sampling of aircraft available for swarm development. However, there are a plethora of airframes in various sizes and configurations both in the hobbyist community and US Department of Defense that could potentially integrate into UAV swarms (Figure 3). One challenge when simulating such UAVs is implementing accurate flight dynamics models (FDM). The FDM used in this thesis is the open source JSBSim software library which is the same FDM used in FlightGear. It has been in constant development by professionals in the fields of aerospace and software engineering since 1996 [5] and relies on aircraft definition files for valid flight characteristic data. The Sig Rascal 110 aircraft model—which has been in widespread use and development in the hobbyist community for nearly 4 years—serves as the airframe used to demonstrate the proposed swarm simulation framework.

Autopilot. As defined by the American Heritage Dictionary, “an autopilot is a navigation mechanism, as on an aircraft, that automatically maintains a preset



Images Courtesy of: <https://images.google.com/>

Figure 3. Diverse Airframes. Small UAVs come in a large variety of shapes, sizes, and configurations. The airframe used in this thesis is the Sig Rascal 110 ARF seen in the top row, second from the left.

course” [36]. In the context of small UAVs, the autopilot does much more. Fulfilling the flight management system (FMS) responsibilities, the autopilot serves as the brain of the UAV and provides supervisory control over other subsystem components. UAV autopilots translate sensor data into output signals that drive the throttle and servos connected to the control surfaces of the UAV, ultimately controlling airspeed, altitude, and heading [39]. Most UAV autopilots provide a minimum of altitude, direction, and position (loiter) hold while some provide the additional functionality of waypoint (i.e. point-to-point) navigation. The autopilot is a critical component of UAVs within a swarm, as it provides stable flight and the ability to navigate. The PX4 flight stack—a Pixhawk autopilot firmware implementation with many onboard applications—is one of many autopilots used in the hobbyist community today and is used in this thesis to demonstrate HIL simulation.

HIL Simulation. Hardware-in-the-Loop is a form of real-time simulation with the addition of a real component in the loop [22] which allows developers to test the performance of real hardware and collect data without taking the risk of losing any real vehicles [7]. This is important when developing resource intensive swarming behaviors because simulated components can perform very different to the actual components due to any assumptions made about the hardware. By integrating the hardware device into the simulation, the physical component will provide the most realistic feedback. Additionally, since the hardware embedded system runs in real-time, the simulation, when utilizing HIL, must also run in real time.

HIL simulation is used profusely through the hobbyist community. This method not only provides hobbyists the ability to practice flight maneuvers with the real hardware autopilot without the risk of crashing the actual UAV, but also allows them to fine-tune autopilot parameters before installing the hardware on an aircraft. The general setup for flying a UAV in the hobbyist community is as follows. An operator uses a radio controller in conjunction with GCS software to communicate with an air vehicle’s autopilot over a communication protocol like MAVLink—a high throughput low-overhead protocol used for communicating with small unmanned vehicles. In manual mode, the operator may provide manual inputs for absolute control over the UAV or switch to another mode, allowing the autopilot to provide partially or fully automated flight; namely, stable flight and point-to-point navigation. HIL simulation replaces the actual UAV with a simulated model. Using this technique, the GCS typically serves as the HIL interface, pulling modeled UAV data (e.g. altitude, attitude, speed, etc) from simulation and sending it to the hardware autopilot in a format the autopilot can understand. In turn, the GCS routes autopilot generated control signals back to the simulation (Figure 4). A major objective of this research effort is to extend this HIL system development approach to support the development of



Figure 4. Hardware-in-the-Loop. The setup on the left shows how small UAVs are typically controlled in the hobbyist community—an operator provides manual RC inputs while passing commands and updating parameters via GCS (both over radio). On the right, the operator controls the simulated vehicle in the same manner. However, the embedded system (i.e. hardware autopilot) communicates with the simulator over a serial/USB connection, with the GCS acting as an interface between the two.

swarm behaviors in simulation that are more likely to transition to real world flight tests with high confidence.

Ground Control Station. A ground control station (GCS) is an interface that provides human operators control of unmanned vehicles in the air or in space. The traditional role of GCSs in UAV operations (especially in the hobbyist community) involves updating UAV autopilot parameters and waypoints while displaying telemetry data received from the UAV [16]. As previously mentioned, GCS software used in the hobbyist community, such as QGroundControl and Mission Planner, also serves as a HIL interface. FlightGear (flight simulator) and QGroundControl are both open source software packages and are used in this thesis to test and develop a hardware interface to the Pixhawk autopilot for HIL simulation in the swarm simulation framework.

2.3.4 Simulation-Chaining.

The architecture of existing flight simulation software could enable a scalable swarm simulation solution called simulation-chaining. XPlane and FlightGear are both flight simulators that already provide realistic flight dynamics modeling of various aircraft types, integrate HIL with existing hardware autopilot flight stacks, and provide network interfacing for simulation input/output (I/O). Simulation-chaining is a term defined here as the integration of multiple flight simulator instances over a network and is described here to point out an alternative approach to providing HIL simulation of UAV swarms and the challenges encountered during its implementation that necessitated the exploration of extending the OpenEagles simulation framework.

The design of simulation-chaining is simple: install existing flight simulation software on two or more computers and add a swarm interface to each, allowing them to communicate across a network (Figure 5). As long as the simulation environment and coordinate system is the same across all flight simulators, individual UAVs can project neighboring UAVs into their environment after receiving information about them over the network. A script could initialize swarm scenarios while a separate tool intercepts/stores/interprets the network traffic (i.e. swarm data) for visualization of the UAVs swarming together. This design requires no further development of flight simulation. Flight simulators such as XPlane and FlightGear allow custom vehicle models, contributing to airframe modularity. They also provide interactive views, though only of a single aircraft, which could prove useful when studying the behavior of each swarming UAV individually. Autopilot HIL implementation is accomplished through 3rd-party GCS software which could integrate into the swarm interface.

To evaluate the feasibility of this solution, a simple application was built to interface multiple instances of FlightGear 3.4.0 (simulating a Sig Rascal 110) and

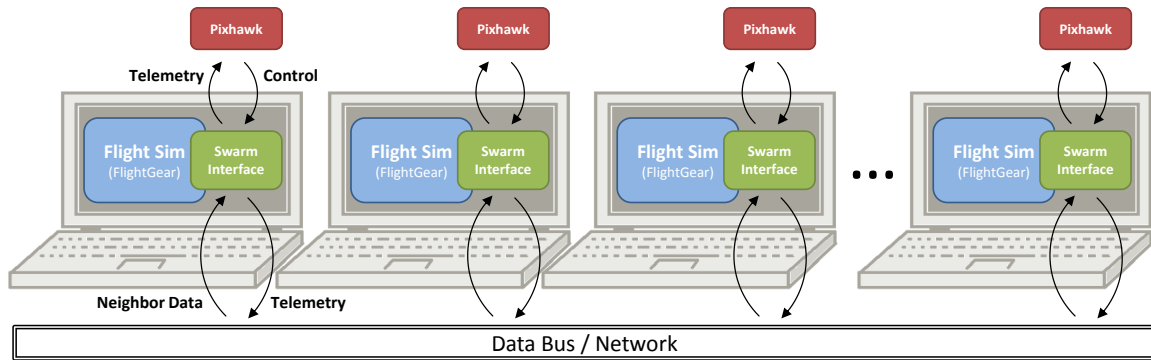


Figure 5. Simulation-Chaining. Each computer runs a unique instance of the simulation, broadcasting its UAV’s simulated telemetry data over a network while simultaneously receiving and processing the telemetry from other simulations.

QGroundControl v2.3.0 over a network. The first challenge in designing the swarm interface was to extract telemetry data from the simulation. An option in FlightGear enabled an output stream of UDP packets containing user specified information about the simulation. This allowed the swarm interfaces to recognize neighboring UAVs. Hardware autopilots (Pixhawks) were connected to the simulators via QGroundControl to enable HIL. In this setup, sets of waypoints programmed into the Pixhawk autopilots provided path planning information (in lieu of actual swarm behavior) which the Pixhawks used to navigate the simulated UAVs. After successfully sim-chaining two simulators together, this setup (Figure 6) successfully enabled two UAVs to recognize each other as if they were flying together in the same virtual environment.

This design accurately models flight dynamics, accommodates HIL, enables interactions among multiple UAVs, and provides a visual interface of the swarm. However, it has a few inherent design flaws. First, the developer must manage each flight simulator instance. This becomes a difficult task when modeling larger numbers of UAVs, especially when initializing complex swarm configurations, resulting in low usability. Also, the implementation of UAV subsystem functionality (e.g. communication and

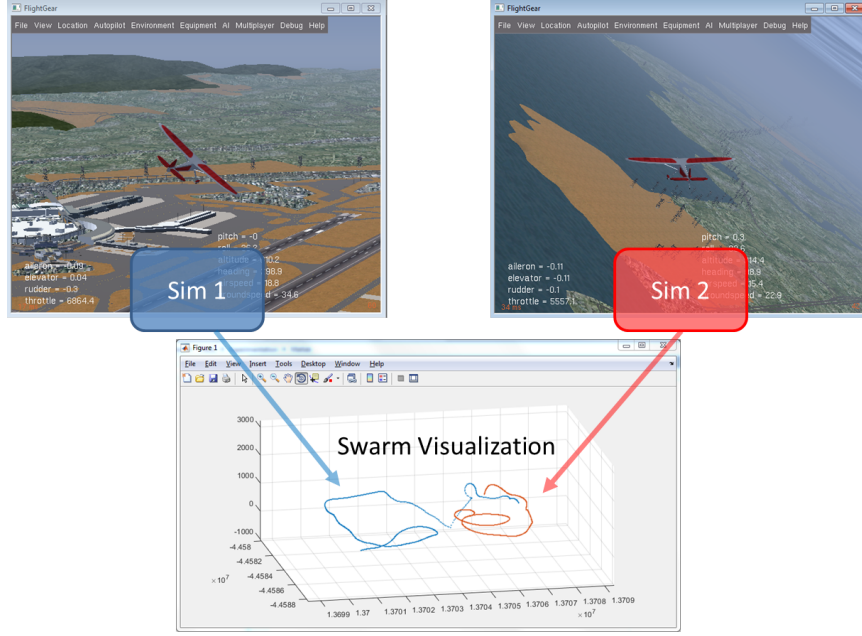


Figure 6. Simulation-Chaining using FlightGear. Two simultaneously executing instances of FlightGear broadcast simulation state via UDP packets. After intercepting these packets, the swarm interface of each instance is able to project its neighboring UAV into its environment as if they are flying together.

sensor systems) must reside in the swarm interface which does not translate gracefully to real aircraft implementation. Lastly, this design does not accommodate the simulation of targets as required in many of the swarm scenarios listed in Table 1. Although simulation-chaining exhibits high potential in many areas, experimentation revealed significant pitfalls necessitating the exploration of an alternative design.

2.4 Quality Attributes

As with any software architecture or design, it is important to explore the relevant software quality attributes. This section focuses on four specific attributes that enhance and promote swarm development: modularity, reusability, usability, and scalability.

During the development process, swarm developers must be able to add, modify, and remove swarm components easily without adversely impacting other parts of the simulation. Therefore, high *modularity*—which is the quality of a system consisting of various parts that separate cleanly and fit together well [35]—should be present throughout the framework. A modular design simplifies testing of various swarm configurations and scenarios, to include different combinations of targets, air vehicles, swarm algorithms, autopilots, and other systems/subsystems.

To enable productive development and compliment framework modularity, swarm simulation components should be reusable. *Reusability* defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time [45]. This quality attribute is important because it promotes developing swarm behaviors over building tools (that enable modeling of swarm behaviors) that may already exist. If custom tools, designs, or documentation are necessary to model a swarm scenario, reusability should be considered to prevent future duplication of work.

In addition to reusability and as required in any software framework, the swarm simulation framework should also exhibit high *usability*, which defines how well the framework meets the requirements of the developer by being intuitive, resulting in a good overall user experience [45]. As seen in the simulation-chaining example from the previous section, if developers find the simulation framework difficult to use or administer, they will abandon it for a more user-friendly alternative. The proposed framework provided tools should be easy to understand and use such that the cost of learning the tools is less than that of pursuing other framework solutions.

Lastly, the framework should be scalable in order to accommodate various swarm sizes and complexities. *Scalability* is the ability of a system to either handle increases

in load without impact on the performance of the system, or the ability to be readily enlarged [45]. The inherent nature of UAV swarms (i.e. strength in numbers) requires high scalability, since low scalability limits what developers can test and experiment with. For example, how does a developer know how a swarm of 200 UAVs will behave if the simulation only has a capacity to model two?

2.5 OpenEaagles

Thus far, we have seen previous approaches, common and essential elements, technologies, techniques, and relevant considerations of simulating UAV swarms. This section introduces the OpenEaagles simulation framework, which this thesis extends to address swarm simulation requirements. The description below, found on the OpenEaagles homepage (www.openeagles.org), summarizes some key distinctions that make OpenEaagles a prime candidate framework foundation:

OpenEaagles is a multi-platform simulation framework targeted to help simulation engineers and software developers rapidly prototype and build robust, scalable, virtual, constructive, stand-alone, and distributed simulation applications... *OpenEaagles* is an acronym that stands for the Open Extensible Architecture for the Analysis and Generation of Linked Simulations. It is a mature software framework as it has been in active development for over a decade. [26]

Additionally, the framework has a modular architecture, enables easy initialization and configuration of simulations that interface with a mature and trusted FDM (JSBSim, the same FDM used in FlightGear), and offers built-in functionalities that promote the configuration of swarms for behavior modeling.

2.5.1 Model View Controller Design Pattern.

OpenEaagles is an object oriented framework, written in C++, and follows the Model View Controller (MVC) design pattern. Each simulation instance is composed

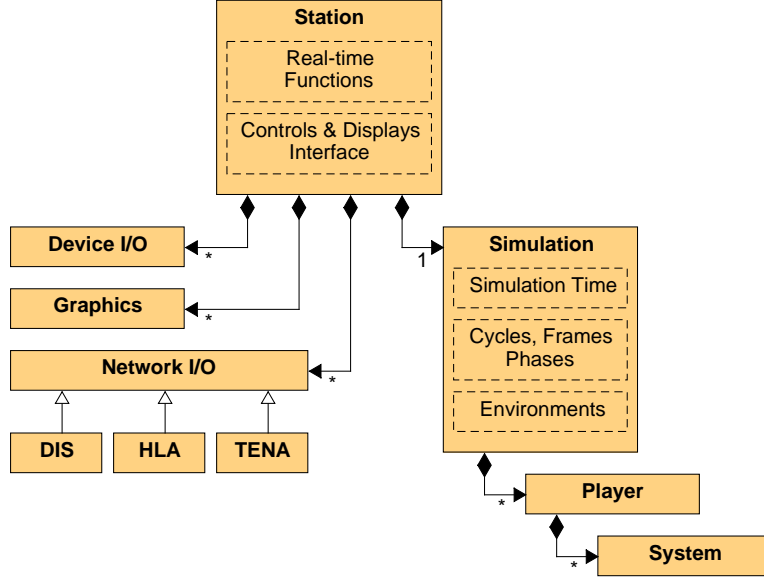


Figure 7. OpenEagles Design Pattern [26]. OpenEagles follows the Model View Controller (MVC) design pattern. The tree of *Players* and *Systems* on the right provides the *model*, the *Graphics* and *I/O* objects on the left provide *view(s)*, and the *Station* and *Simulation* both serve as *controllers*.

of various objects in a tree data structure (Figure 7). The model comprises *Player* and *System* objects, a variety of optional graphics and I/O interfaces provides customized view(s), and a *Station* object sits at the root of the tree and interfaces with the *Simulation* object to fulfill simulation controller functions.

Model. Each simulation instance has one Simulation object. The Simulation has *Players*, and *Players* have *Systems*. *Players* represent the entities modeled (e.g. UAVs and targets) while the *Systems* represent devices, utilities, and tools (e.g. radios, dynamics models, sensors) used by the entities. Developers can subclass each *Player* and *System* class for customization. Figure 8 shows some UAV systems relevant to swarms and how they can be subclassed in OpenEagles. This simple design, using both inheritance and polymorphism, allows for modeling of unlimited swarm configurations.

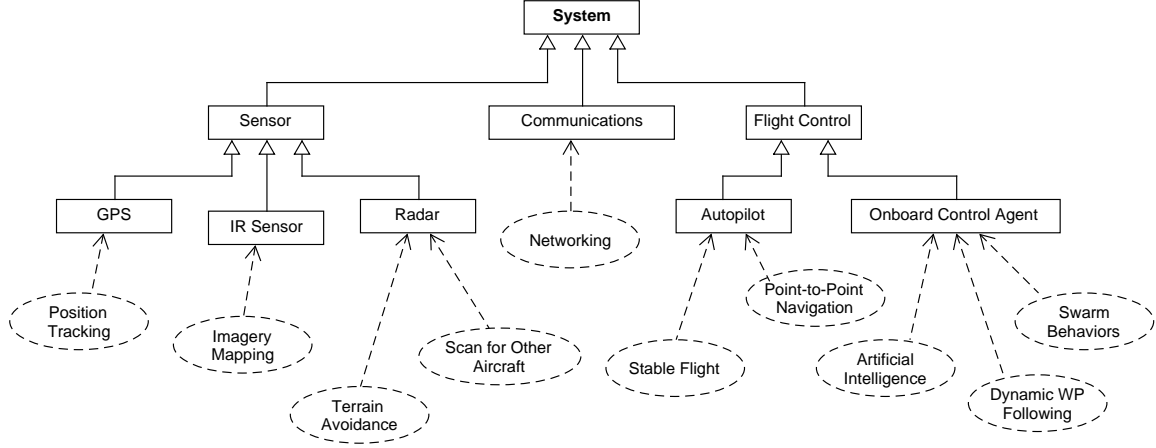


Figure 8. UAV System Hierarchy. UAV systems and subsystems can be grouped by functional role (circled beneath). In OpenEagles, these systems are implemented with parent/child relationships using inheritance to provide benefits such as scalability and code reusability.

Furthermore, each System has a method called *getOwnership* which returns a pointer to the owning Player. Players have the method *getSimulation* and Simulations have the method *getStation*. Since the Station object has access to the entire tree and each object can reach back to the Station, any object can access any other object directly. This gives Players and Systems the ability to interact with each other.

View. To view simulation data, OpenEagles includes a graphics package that leverages OpenGL. With this, developers can design and implement custom views showing various information about Player(s) such as their attitude, airspeed, fuel consumption, location on a map, or a visual of surrounding terrain. The framework also contains packages that enable networking over distributed simulation architectures such as Distributed Interactive Simulation (DIS), High-Level Architecture (HLA), and Test and Training Enabling Architecture (TENA). These packages meet a wide range of custom visualization requirements without the overhead of developing custom tools.

Controller. As mentioned previously, the Station and Simulation objects provide simulation control. The Station interfaces with devices, graphics, and network

objects and provides control over refresh rates, priorities, and thread stack sizes. The *Simulation* controls simulation specific parameters (e.g. reference points, time of day, Earth models, etc.) and provides thread management over both time-critical and non-time-critical tasks. Specifically, the *Simulation* object allows developers to specify the number of time-critical and background threads which are assigned *Players* in round-robin fashion such that each thread processes a subset of *Players*. Time-critical thread(s) are reserved for performing tasks that must remain synchronized with real-time while the non-time-critical thread(s) allow other tasks to run in the background without blocking or interfering with time-critical tasks. The distinction between the two is especially important when modeling in real-time, such as when hardware autopilots are used in HIL simulation. Because the hardware autopilots execute in real-time, the simulation must also execute in real-time to provide corresponding simulation data in sync with the real-time control signals received from the autopilot. Additionally, multithreading in OpenEaagles can be useful when simulating large numbers of *Players* because threads are distributed across multiple CPU cores (if present) and process *Player* tasks concurrently. However, such parallel processing does require overhead that is less advantageous when modeling fewer *Players*.

2.5.2 Base Classes.

At the foundation of OpenEaagles are its base classes (Figure 9). All objects are of type *Object* which contains the methods *ref* and *unref* used for memory management. Each *Component* object may hold other components (i.e. subcomponents) in a *PairStream* list. Each OpenEaagles tree data structure takes advantage of this component/subcomponent relationship, giving the framework its scalability. The tree is updated via *updateTC* and *updateData* methods, corresponding to the previously mentioned time-critical and non-time-critical threads respectively. Specifically, these

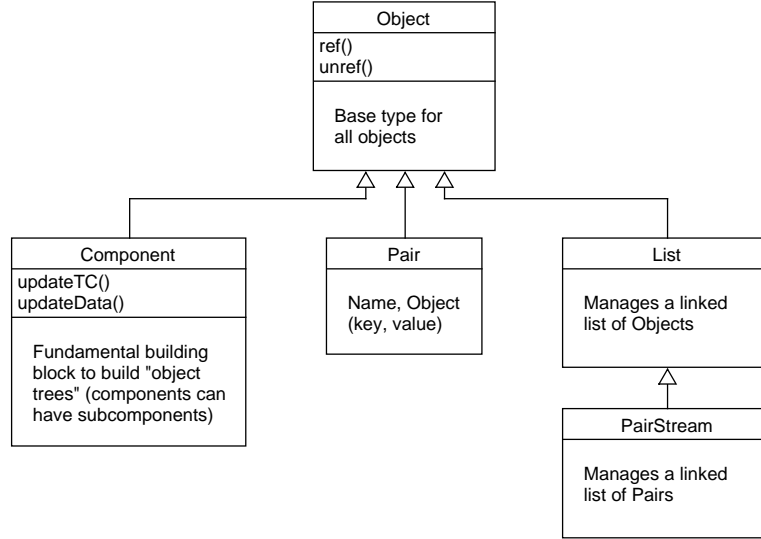


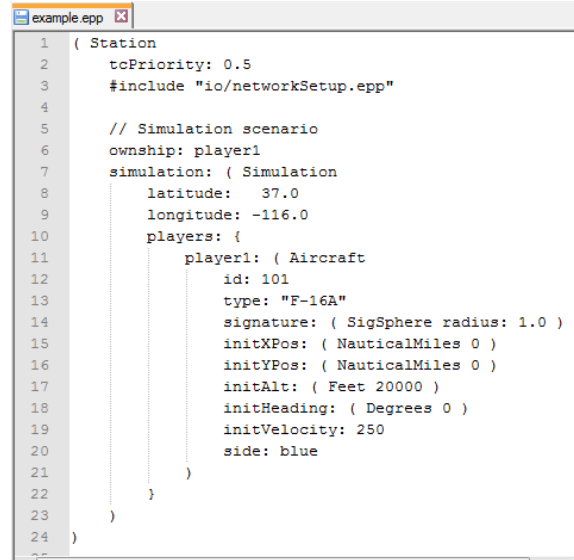
Figure 9. OpenEagles Base Classes [26]. All objects in OpenEagles are of type *Object*. Some objects are also *Component* objects which hold a *PairStream* of other components, enabling a tree structure of subcomponents.

update methods are called upon at the root of the simulation tree. The root's update methods in turn call upon corresponding update methods of any subcomponents, which subsequently calls the update methods of their subcomponents until the entire tree has been updated. Updates reoccur at the specified refresh rate.

Simulation synchronization is accomplished through the use of frames. A *frame* in OpenEagles is defined as the period in which all updateTC methods in the simulation tree are called. Time-critical tasks from a single frame must not begin until after the completion of all time-critical tasks from the previous frame. Additionally, the minimum duration of each frame is defined by the specified frame rate (e.g. 20 ms assuming a 50 Hz refresh rate).

2.5.3 Eaagles Definition Language.

In addition to understanding the basic building blocks and structure of OpenEagles, its use requires familiarity with the Eaagles Definition Language (EDL) which is used to initialize and configure each simulation and gives users complete control over the



```

1 ( Station
2   tcPriority: 0.5
3   #include "io/networkSetup.epp"
4
5   // Simulation scenario
6   ownship: player1
7   simulation: ( Simulation
8     latitude: 37.0
9     longitude: -116.0
10    players: {
11      player1: ( Aircraft
12        id: 101
13        type: "F-16A"
14        signature: ( SigSphere radius: 1.0 )
15        initXPos: ( NauticalMiles 0 )
16        initYPos: ( NauticalMiles 0 )
17        initAlt: ( Feet 20000 )
18        initHeading: ( Degrees 0 )
19        initVelocity: 250
20        side: blue
21      )
22    }
23  )
24 )

```

Figure 10. Example EPP File. This simple text file provides developers the ability to specify reusable swarm configurations. This particular definition includes network settings imported from the networkSetup.epp file using the “#include” syntax and defines a simulation with a single player entity—an Aircraft of type F-16A.

initial size, configuration, and state of the simulation tree as well as its refresh rate. EDL files allow developers to dynamically customize the initial simulation state. With a simple plaintext file, users can define objects and their corresponding attributes. This includes the relationship between components and subcomponents. Developers can either define objects and/or attributes directly in the EDL file, or decompose configurations into Eaagles Pre-Processed (EPP) subfiles—a *makeEdl* batch script in turn merges the EPP files together into a single EDL file. This decomposition technique is powerful in that it allows reuse of configuration settings.

The example EPP file shown in Figure 10 demonstrates how a simulation tree is defined. This particular tree, after OpenEaagles parses it, will hold three top level objects: a Station, Simulation, and Aircraft. As you can see, a number of attributes and initial conditions are also defined, such as the origin of the simulation (37.0°N, -116.0°W) as well as the player’s ID (101), aircraft type (F-16A), initial offset from the origin, etc. Network configurations are defined in a separate EPP file,

namely *networkSetup.epp* found in the *io* directory. As seen in this example, using the *#include* syntax enables reusability—“write once, use many”. For a comprehensive description of the Eaagles Definition Language, see *Basic Package Classes Slides* under the Documentation section of the OpenEaagles website [26].

2.5.4 Main Methods.

The strength of OpenEaagles comes from its existing building blocks. With the exception of the Main class, a fully operational multi-agent flight simulation in OpenEaagles requires no additional source code. Existing player, dynamics model, and system classes provide everything a developer needs to build complex large-scale models of UAV formations. However, the Main class requires the implementation of three specific methods before it can translate EDL files into simulation trees and update them at specified refresh rates. Developers must implement the *createObj*, *builder*, and *main* methods before executing any simulation. Figure 11 shows the parsing process used by these methods.

The *createObj* method takes in, as a parameter, a character string. If the string matches the name of one of the defined objects, this method creates an instance of that object and returns it to the parser. In turn, the parse applies any attributes specified in the EDL file using “slots” defined in the object’s class. The *createObj* method enables fine control over which objects are available for parsing and how the parser will instantiate them.

Next, developers must implement the *builder* method by passing the *createObj* method as well as the EDL file name to the parser. A pointer to the Station object of the simulation tree is returned by the parse which the *builder* method passes to the *main* method. Lastly, developers must implement the *main* method such that it iterates simulation updates as needed. For HIL simulation, this method must execute

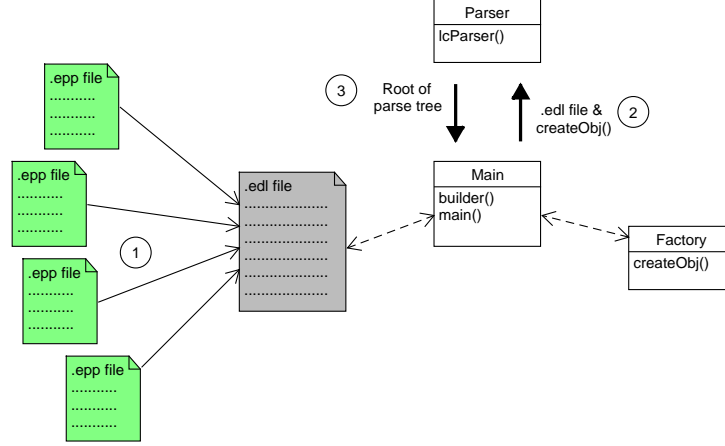


Figure 11. Eaagles Definition Language Parsing. 1) A batch script converts Eaagles Preprocessed (.epp) files into a single Eaagles Definition Language (.edl) file. 2) The *builder* method in the Main class calls upon the *lcParser* method by passing it the file name of the .edl file as well as the *createObj* method from the Factory class as parameters. 3) In turn, the Parser uses the *createObj* method to convert the .edl file into a simulation parse tree. A pointer to the root of this tree is returned to the main method for simulation execution.

in real-time, providing periodic updates in sync with the system clock time and at a refresh rate matching that which is required by the hardware embedded system. The implemented logic in the *main* method should account for *waits* or *delays* necessary to align “simulation time” with “real-time”.

2.5.5 Extending Built-in Functionality.

OpenEaagles contains many packages with built-in functionality that already satisfies a variety of swarm simulation requirements (Figure 12). By extending these packages, developers can implement custom vehicles, sensors, flight dynamics, and autopilots. For example, a developer could subclass the already existing *AirVehicle* class into an *UnmannedAirVehicle* class that contains subsystems specific to autonomous aircraft. This grants great flexibility in modeled different swarm configurations.

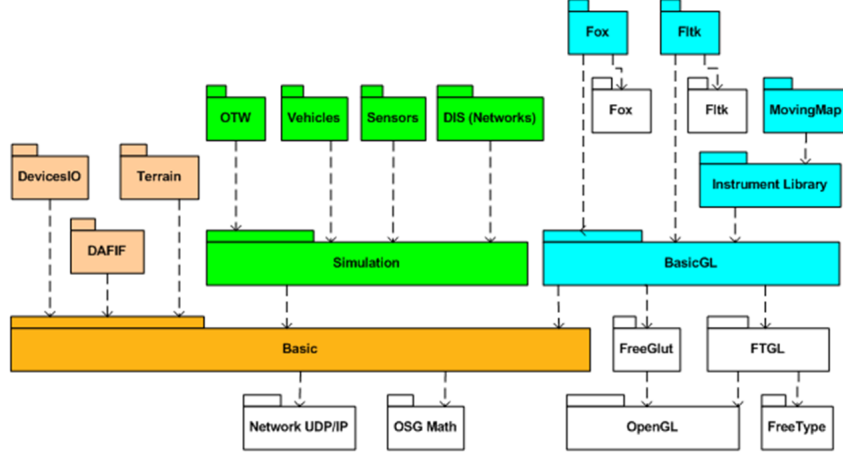


Figure 12. OpenEagles Package Hierarchy. Packages with a white/clear background indicate the use of 3rd party open source tools. (Originally presented in [26] and reproduced as an element of the public domain.)

Three classes within these packages that are specifically relevant to swarm development efforts include the *AirVehicle*, *Pilot*, and *AerodynamicsModel*. The *AirVehicle* is a subclass of type *Player* that holds subsystems. Again, developers can extend it to a UAV subclass, implementing functionality specific to UAVs in a swarm. The *Pilot* is a generic object that provides control (e.g. throttle position) over a *Player* object. In the context of swarm modeling, the *Pilot* can be extended to an *Autopilot* subclass, implementing controls specific to the control surfaces of the modeled aircraft. The *AerodynamicsModel* is the *AirVehicle* System that provides the aircraft its flight characteristics. In other words, the *AerodynamicsModel* defines what “kind” of aircraft the *AirVehicle* will model. By extending the *AerodynamicsModel*, developers can use built-in, 3rd-party, or custom FDMs. OpenEagles already contains a JSBSim interface, called *JSBSimModel*, that extends the *AerodynamicsModel* class. Since JSBSim provides realistic flight modeling, *Players* that use this FDM will inherit such realistic flight behavior.

2.6 Summary

UAV swarms show tremendous potential. To realize this potential, swarm developers need a tool that allows them to accurately model and test their swarm theories, algorithms and control strategies using hardware-in-the-loop. Although previous swarm simulation approaches fell short of satisfying such need, commonalities among them revealed essential simulation components necessary for any swarm simulation framework. Additionally, many tools and techniques developed in the hobbyist community prove highly relevant to the design of a successful swarm simulation framework. An implementation of simulation-chaining showed one approach that applies tools from the hobbyist community toward simulating swarms, but was found to be fraught with inherent design flaws which resulted in poor usability and modularity. Finally, OpenEagles was introduced as a modular and scalable solution which is extended to create the HIL capable swarm simulation framework introduced next.

III. Framework Design and Analysis

The OpenEagles simulation framework was chosen for this effort because of how well it met requirements explored in the previous chapter. Those requirements continue to guide an extension of OpenEagles here to accommodate the simulation of swarms using hardware-in-the-loop (HIL). This chapter is organized as follows. First, a swarm simulation component serving as the sandbox for swarm development is defined. Next, the OpenEagles extension (i.e. proposed swarm simulation framework) is described. Lastly, the framework assessments portion outlines assumptions, limitations, and intended measures of framework demonstrations, followed by the scenario use in the demonstrations.

3.1 Swarm Algorithm Placement

Before extending OpenEagles to enable swarm behaviors, the space for swarm algorithm placement must be defined. Should swarm algorithms be integrated into existing UAV systems or confined to a standalone component that interfaces with such systems? This section first defines a functionally isolated unit that encapsulates swarm algorithms, then argues where this unit should reside.

Onboard Control Agent. During a traditional UAV mission, a ground control station provides operators the ability to pass path planning information to the autopilot which directs it where to fly (Figure 13). However, the nature of swarming UAVs requires autonomous path planning without operator input—individual UAVs must act autonomously. This is possible when each UAV has an intelligent agent onboard for independent decision making (i.e. without external intervention). Agents are considered intelligent and capable of autonomous behavior if they are proactive (goal-directed behavior), reactive (respond to change), and have social abilities (interact

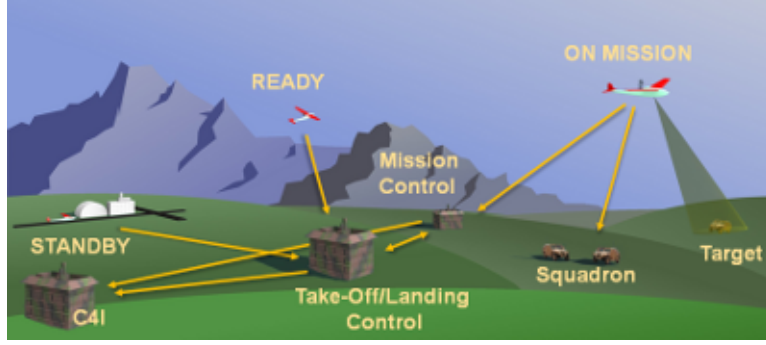


Figure 13. UAV Mission Overview. This graphic depicts a typical UAV mission where a GCS (shown here as Mission Control) communicates mission parameters and guidance information to UAVs in the field while in return receiving sensor information captured during the mission. (Originally presented in [10] and reproduced as an element of the public domain.)

with other agents) [46]. The term Onboard Control Agent (OCA) is used in this thesis to refer to the functional component that provides such capability.

To implement swarm behavior in both simulation and real aircraft, the OCA must have a thread to execute on, access to data necessary for autonomous decision making, and the ability to provide the autopilot with path planning information. To satisfy these requirements, only two locations make sense. The OCA may reside within the autopilot integrated with the flight management system (FMS) or external to the autopilot as a standalone component. Considering the differences between these two approaches is especially important when implementing a hardware-in-the-loop (HIL) simulation framework.

Integrated OCA. Since the autopilot already has access to sensor data and is capable of executing code, it serves as a natural first choice for swarm algorithm placement. Furthermore, with the OCA and autopilot integrated, path planning information is generated and communicated internally, thus providing some inherent benefits.

First, the autopilot and OCA often require similar outer-loop sensory information to perform their corresponding functional roles. Integrating them together in the

same component (i.e. software class) simplifies data management and obviates duplicate information shared between two separate components. By integrating them together, the OCA can interact with the autopilot FMS by calling upon attributes and methods directly, instead of implementing interfaces between separate components. Additionally, this approach simplifies the transition from HIL simulation to real flight testing because the OCA already resides in the autopilot hardware and does not require implementation on a separate hardware device.

Unfortunately, from a software development perspective, these benefits come with significant drawbacks. Co-locating the autopilot and OCA induces tight coupling between the two which reduces modularity—for example, swapping to a different autopilot design or platform would require an OCA implementation specific to the new autopilot. Furthermore, integrating OCA functionality into the autopilot requires high understanding/familiarity of autopilot implementation details. Instead of simply interfacing with an autopilot (i.e. passing it waypoints in exchange for controlled flight through those waypoints), developers must modify the autopilot code base or firmware (for HIL) to house the OCA without impacting FMS functionality.

In addition to these implementation specific shortfalls, another setback to such an integrated approach involves resource utilization. The CPU(s) of an autopilot is a finite resource already dedicated to specific tasks (e.g. providing stable flight and point-to-point navigation). When sharing a processor, the autopilot and OCA must compete for execution time. Computationally heavy swarm algorithms may consume enough CPU cycles to starve or disrupt autopilot FMS execution, thus risking forfeiture of stable flight.

Standalone OCA. An alternative approach is to separate OCA functionality into its own component. This approach alleviates the drawbacks of the integrated approach because the autopilot and OCA are separate components that no longer

compete for the same CPU cycles. Such an approach requires that we encapsulate functionality between the two components, thus increasing modularity. With only a simple interface, various OCA implementations can pair with different autopilots without requiring significant changes—developers can now add swarm algorithms without impacting FMS execution or learning complex autopilot implementation details.

While some trade-offs are made for these advantages, they are relatively minor. Consider a HIL simulation where the OCA is implemented on a dedicated hardware device, such as a Raspberry Pi. Combining the autopilot with such an OCA would require the implementation of an interface or communication protocol between the two. Fortunately, many low-overhead high-throughput protocols currently exist that would satisfy this requirement. Ultimately, this modular approach is adopted as the approach presented in this work.

3.2 Communication Flow

Thus far, we have established three specific simulated UAV systems required for swarm development: *flight dynamics models (FDM)* to define UAV flight characteristics and motion through the simulated environment, *autopilots* for stable flight and navigation, and *OCAs* for swarm algorithm encapsulation. Additionally, the *visual interface* to the simulation must have access to aircraft information generated by the FDMs to provide a 3D view of swarm behaviors. Determining the flow of information between these simulation components helps define how they will fit together in the framework structure.

The functional roles of these four simulation components dictate how they will interact and what information they will share. The OCA has the primary role of autonomous decision making and path planning—based on its reactivity, proactivity,

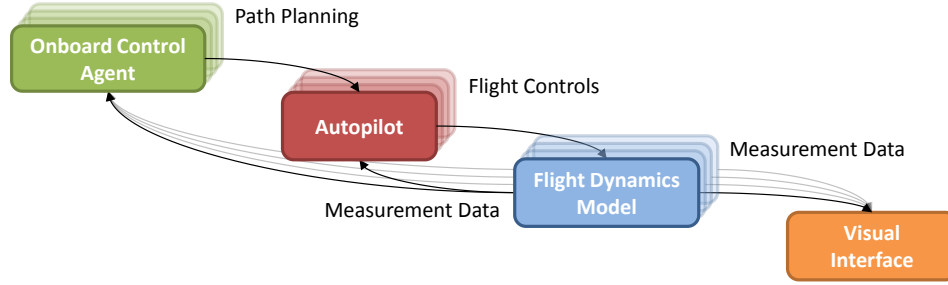


Figure 14. Communication Diagram of Swarm Simulation Components. Swarm behavior originates in the OCAs and manifests as path planning information derived from measurement data processed by swarm algorithms. Autopilots translate the path information from the OCAs and measurement data received from the FDMs into flight control signals, forming a feedback loop. Simulation views are subsequently generated by interpreting measurement data from the FDMs. Note that each OCA requires measurement data from multiple FDMs to accommodate its social abilities.

and social ability—which requires an input of sensor and communication data. The role of the autopilot is to provide flight control signals to the aircraft for stable flight and navigation, which requires path planning information and a feedback loop of aircraft measurement data. The role of the FDM is to model aircraft measurement data as time progresses and flight control signals are received. Lastly, the visual interface to the simulation requires aircraft measurement data. Matching the required component inputs to corresponding outputs results in the feedback loops and flow of information necessary for swarm behavior modeling. This communications flow, summarized in Figure 14, is implemented in the swarm simulation framework design.

3.3 Extending OpenEagles

Recall from the previous chapter how OpenEagles enables scalable and modular multi-entity simulations. This section describes the extensions required to enable swarm behavior modeling and HIL simulation.

3.3.1 Enabling Swarm Behaviors.

As previously described, the autopilot provides stable flight and point-to-point navigation while the OCA provides autonomous decision making. But how do they work together to enable swarm behaviors? Within a swarm, the OCA embedded in each simulated UAV intelligently performs path planning “on the fly.” This means the destination path of the UAV is constantly changing based on the proactivity, reactivity, and social abilities of the OCA. The medium for communicating this dynamic path information to the autopilot are pointing vectors, which are translated into sets of waypoints. In other words, autopilots are capable of autonomously navigating through preset waypoints, and by dynamically updating these waypoints, the OCA gains control of where the autopilot navigates to. Furthermore, because the path is constantly changing, the autopilot only requires a single dynamic waypoint to follow, like “chasing a carrot on a stick.” Using such dynamic waypoint following (DWF) technique, the OCA controls the autopilot and enables swarm behaviors.

In OpenEagles, the autopilot and OCA are subsystems of the UAV player entity. To accommodate DWF, autopilot classes must have methods that set and update the dynamic waypoint. Therefore, a top-level abstract class containing these methods is necessary. Such a class enforces DWF in any autopilot subclass while allowing custom autopilot design at the base class level, thus retaining autopilot modularity. Figure 15 shows the three subsystems (FDM, OCA, and autopilot) of a UAV that enable the simulation of swarm behaviors. The FDM models the aircraft, accepting flight control inputs while producing simulated remote measurement/telemetry data. The OCA houses the swarm algorithms necessary for intelligent autonomous behavior—this is where swarm developers will spend most of their time. With sensor inputs from other UAV subsystems (not shown in Figure 15) as well as inputs from other neighboring UAVs, the OCA can make decisions and perform path planning

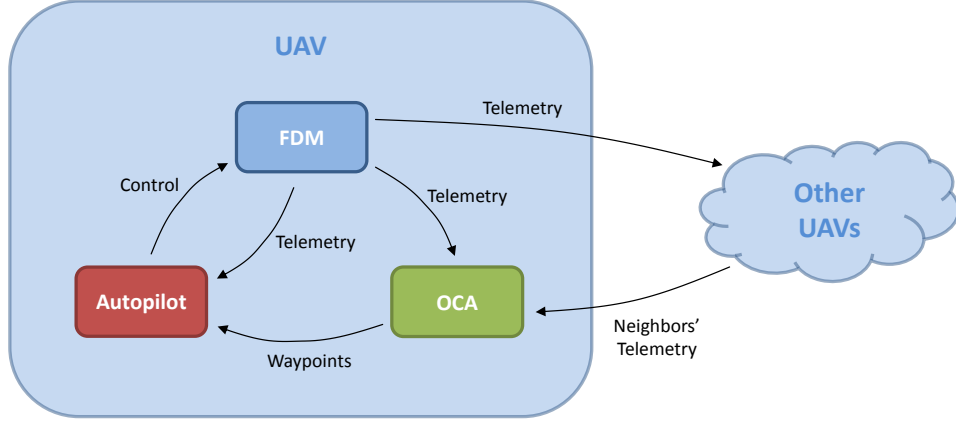


Figure 15. UAV Systems Communication Diagram. In swarm simulations, the OCA receives measurement data (i.e. telemetry) from the FDM of its corresponding UAV as well as data from neighboring UAVs. Swarm algorithms implemented in the OCA translate this data into autonomous path planning information passed to the autopilot in the form of waypoints (i.e. dynamic waypoint). Subsequently, the autopilot continuously navigates the simulated aircraft toward the dynamic waypoint. These interactions result in simulated swarm behaviors.

independent from external intervention, thus enabling swarm behaviors. Lastly, the autopilot continuously accepts path planning information in the form of a dynamic waypoint from the OCA and drives the FDM to fly the modeled aircraft toward that waypoint via flight control signals.

3.3.2 Enabling Hardware-in-the-Loop.

To accommodate HIL simulation, OpenEagles must interface with hardware devices. To maintain the design pattern and system/subsystem relationships used in OpenEagles, the system objects corresponding to the hardware devices act as hardware interfaces instead of functional components (Figure 16). Such interfaces manage serial connections to the devices as well as translate and route signals between the simulation and the device. For example, the *Autopilot* hardware interface object continues to communicate with the FDM and OCA as before, but instead of converting input (i.e. telemetry and waypoints) directly into output (i.e. flight control signals), it passes that input to a hardware device (such as a Pixhawk) in a format the

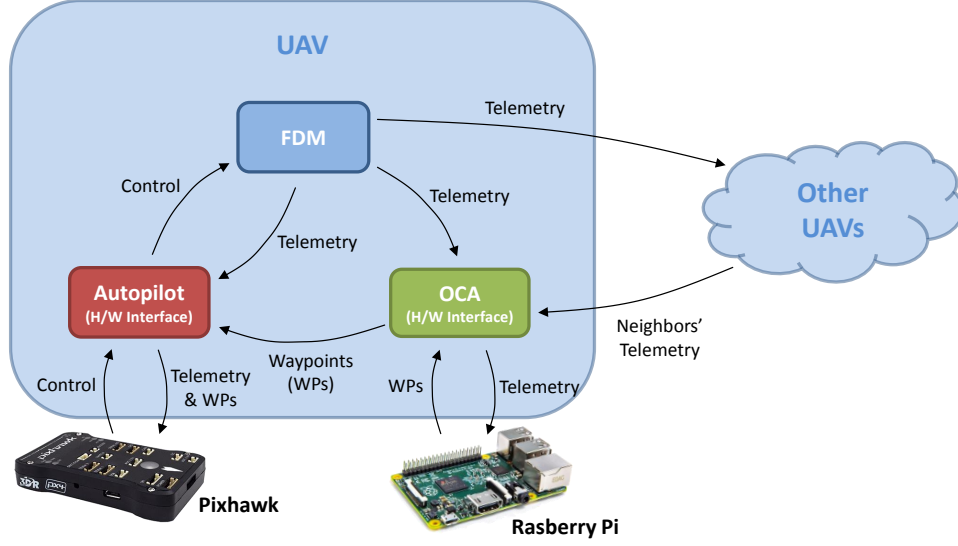


Figure 16. UAV Systems Communication Diagram (with HIL). When implementing HIL simulation, hardware devices provide system functionality directly to the simulation via hardware interfaces. In this example, a Pixhawk and Raspberry Pi provide autopilot and OCA functionality respectively while the *Autopilot* and *OCA* system objects serve as interfaces to those hardware devices. (The Raspberry Pi is shown here for demonstration purposes only. No OCA hardware interface is implemented in this thesis.)

device can understand. The device performs the autopilot behavior with the given input signals and responds with flight control commands that the *Autopilot* object translates to FDM-readable control signals.

Some challenges associated with serial connection management include the accommodation of high baud rates (required in HIL simulation) and asynchronous communication with hardware devices. HIL simulation requires real-time execution at specific refresh rates (e.g. Pixhawk PX4 flight stack requires HIL_SENSOR and HIL_GPS packet updates at 50Hz and 10Hz respectively). The time-critical thread must update the entire simulation tree within each interval period of the highest refresh rate to provide valid data. Additionally, send and receive streams from the hardware device communicate data in parallel and at different rates, which requires careful management of serial traffic.

3.3.3 The Swarm Simulation Framework.

Swarm behavior modeling using HIL simulation is accomplished by extending OpenEagles as shown in Figure 17. The OpenEagles default *AirVehicle* class contains slots for specifying a pilot and dynamics model. Therefore, any subclass of *AirVehicle* may contain such systems or subclasses of those systems (e.g. autopilot and *JSBSimModel*). The UAV subclass is necessary to uniquely identify UAVs from other *AirVehicle* entities during simulation run-time and implement an additional slot for specifying the OCA.

From such a design, UAV players now contain all three systems necessary to accommodate swarm behaviors: *SwarmAutopilot* (autopilot), *JSBSimModel* (FDM), and *OnboardControlAgent* (OCA). Each autopilot class extends from a *SwarmAutopilot* parent class that enforces DWF functionality. *SwarmAutopilot* subclasses may either provide autopilot functionality directly (fully simulated) or serve as an interface to corresponding hardware devices (HIL). By subclassing the *SwarmAutopilot* class and providing a corresponding hardware interface, developers can integrate any hardware autopilot (e.g. ArduPilotMega, OpenPilot, Armazila, etc.). Similarly, swarm algorithm developers can implement swarm behaviors in *OnboardControlAgent* objects or provide the interface to corresponding hardware devices (e.g. Raspberry Pi) where development takes place. For this effort, *PixhawkAP* will serve as the hardware interface to the Pixhawk autopilot.

Using public methods, the OCA pulls telemetry directly from the FDM and sets the dynamic waypoint of the autopilot while the autopilot sets the controls (i.e. rudder, stick, throttle) of the FDM. The final swarm simulation framework design provides a modular, scalable, and accurate means of comprehensive swarm behavior modeling while accommodating HIL simulation as a means of approaching an actual operational configuration.

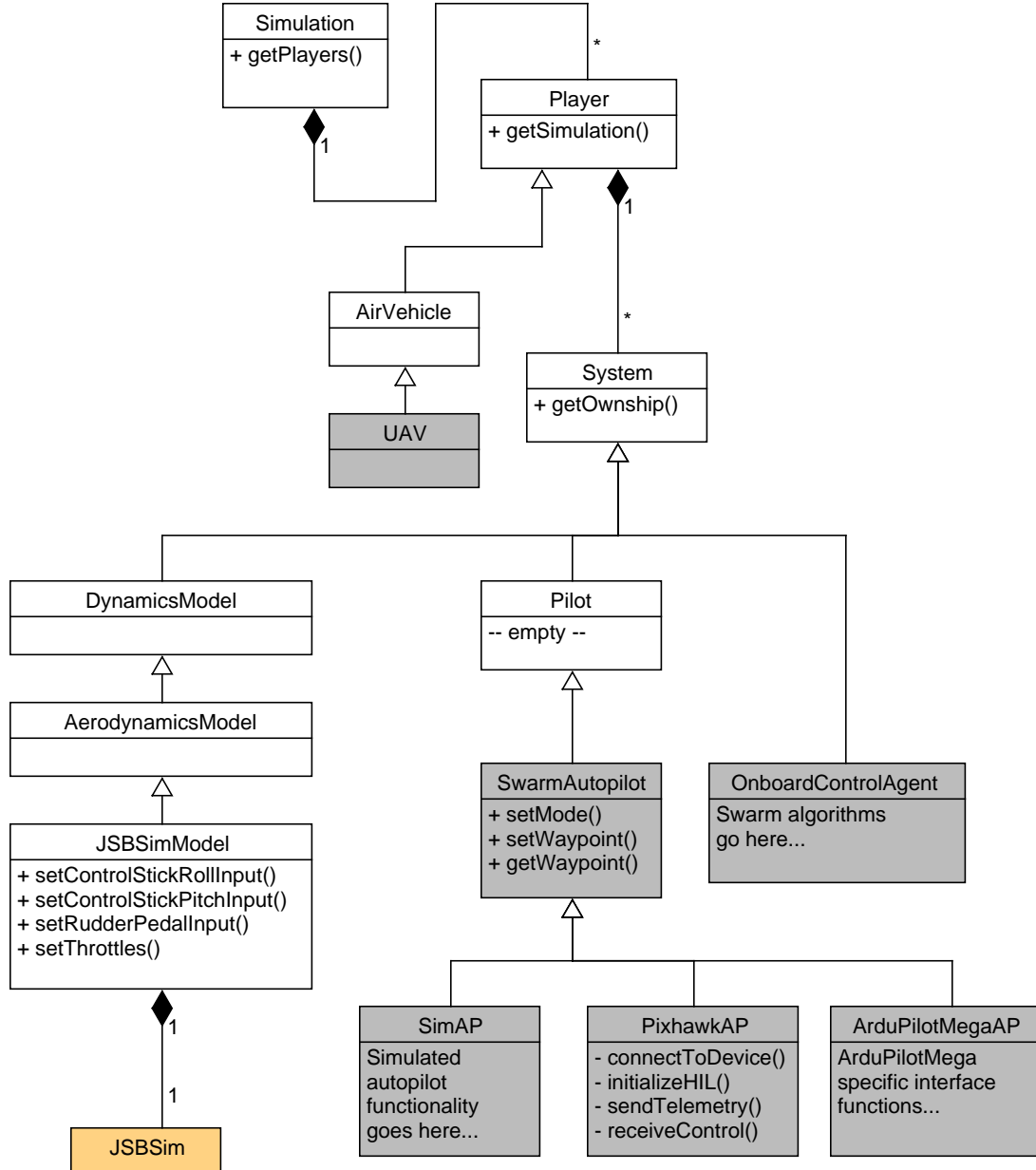


Figure 17. Swarm Framework Class Diagram. OpenEagles (shown in white) is the foundation for the proposed swarm simulation framework (shown in gray). All *Player* subclasses have *DynamicsModel* and *Pilot* subsystems. The *AirVehicle* subclass, *UAV*, is necessary to accommodate an *OnboardControlAgent* subsystem for swarm behavior implementation. The *OnboardControlAgent* encapsulates swarm behavior development and imposes DWF by pushing waypoints to the *SwarmAutopilot*. *SwarmAutopilot* subclasses either implement a simulated autopilot or serve as an interface to a hardware autopilot during HIL simulation. As show at the bottom left, *JSBSimModel* is the built-in OpenEagles interface to the popular open source flight dynamics model, *JSBSim*.

3.4 Framework Assessments

Both qualitative and quantitative analysis are used to assess how the framework accommodates realistic modular and scalable UAV swarms in HIL simulation. This section outlines assumptions and limitations of the analysis performed, specific focus areas of the analysis, and the swarm development scenario wherein the framework capabilities will be demonstrated.

3.4.1 Assumptions and Limitations.

UAVs are typically categorized as either fixed-wing aircraft or rotorcraft, each of which have distinctive characteristics that pose separate challenges in designing autonomous behavior [4]. Although the swarm simulation framework proposed here can extend to both categories in theory, this thesis focuses exclusively on small fixed-wing UAVs with wingspans of less than 10 feet. Currently, this framework does not account for weather related effects such as precipitation and wind or gusty flight conditions. Although the framework can accommodate many different aircraft models, autopilots (which can now be either purely simulated or interfaces to hardware embedded systems), and swarming algorithms, the demonstrations in the next chapter implement swarm behaviors with only the following configurations:

- *Aircraft model*: unofficial (i.e. untested) model of the Sig Rascal 110 airframe found on github.com [18]
- *Simulated autopilot*: custom software autopilot implemented and tuned to control the Rascal model—stick and rudder controls (with constant 100% throttle) provide heading and altitude holds when given a waypoint or set of waypoints

- *HIL autopilot*: Pixhawk with PX4 firmware (px4fmu-v2.default.px4) set to control the Rascal model using default settings provided in QGroundControl during device initialization/setup
- *Swarm algorithm*: implementation of Reynolds Flocking rules using separation, alignment, and cohesion vectoring as detailed by Brundage [6]

Limitations of scalability are measured for the above configurations. However, these measurements are restricted to the platform used in this effort—an HP Elite-Book 8560w with a 2.20 GHz Intel® Core™ i7-2670QM CPU, 16.0 GB of RAM, and a 64-bit operating system (Windows 7 Professional with Service Pack 1). Additionally, the scope of this thesis will not cover virtual or physical clustering as a means of scaling swarm simulations beyond the capabilities of a single platform. Instead, simulations will use multithreading with seven time-critical threads and one background thread distributed across eight CPU cores. Time-critical threads will process FDM and autopilot tasks while the background thread will process OCA tasks.

As previously discussed, some UAV configurations have complex communication systems and onboard sensor packages requiring simulation. Although the proposed framework can accommodate these complexities, the final proof of concept implemented here will not apply such communications and sensor processing, but instead assume entities within the simulation have direct access (handles/pointers) to instances of other entities, calling upon their publically available attributes directly.

3.4.2 Qualitative Measures.

The ability to easily add and remove components within a swarm during the development process is crucial to effective and efficient swarm development. As swarm components are added, modified, and removed from the simulation tree, the framework will be assessed on ease of integration, interoperability between simulation components, and encapsulation of swarm behaviors.

To provide realistic simulations of swarm behaviors with high confidence those behaviors will transition to real flight tests, the flight dynamics model (i.e. JSBSim) used should model accurate aircraft flight characteristics. Observations on aircraft performance—including cruise airspeed, stall speed, responsiveness, and the accomplishment of various maneuvers—should provide strong indications of the accuracy of aircraft behavior modeling and ultimately increase confidence on swarm behavior accuracy. For example, if a real aircraft is rated with a maximum airspeed of 60 knots when flying straight and level, a similar simulated flight of the same aircraft should result in similar airspeeds. Qualitative analysis of individual aircraft behaviors will be assessed by providing specific control inputs (e.g. right stick) and comparing resulting behaviors to that of expected outcomes (e.g. roll to the right).

3.4.3 Quantitative Measures.

Having shown how the proposed framework achieves scalability, we can now assess the quality attribute during HIL simulation. Because hardware devices external to the simulation execute in real-time, the simulation must do the same or risk invalidating the simulation. However, as the swarm size grows, so does the size of the simulation tree and consequently the quantity of operations required (within a specific time frame) to update and advance the simulation in real-time. To assess framework scalability during HIL simulation, two performance metrics will be used: 1) duration

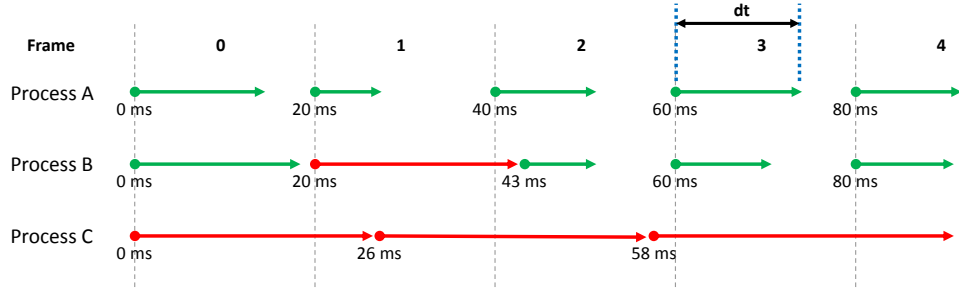


Figure 18. Simulation Run-time Graph. Each of the three processes require a 50 Hz refresh rate with 20 ms frames. Arrows indicate the duration of each simulation tree update. Green updates finish before the end of their current frame and thus enter a brief wait until the next frame begins while red updates do not.

of simulation tree updates measured in milliseconds to indicate whether real-time is achieved, and 2) maximum swarm size (i.e. UAV count) to indicate overall limitations of framework scalability.

Update Duration. The refresh rate, also known as the *frame rate*, of the simulation determines the minimum duration between the start of each simulation tree update. When the duration of an update, dt , is less than the duration of the frame, the simulation waits until the end of the current frame before preceding to the next and is able to remain synchronized with real-time. However, when dt is greater than the duration of the frame, the next update will not begin until the current one finishes which causes the simulation to drift away from real-time synchronization.

Figure 18 shows three example processes, each executing a different simulation. Process A always finishes updating the simulation tree prior to the end of each frame, thus entering a brief wait between every update. The second update for Process B extends past its corresponding frame, but the duration of the following update is short enough to recovers real-time execution. For Process C, all updates exceed the frame duration, wherein the cumulative effect causes the simulation to diverge away from real-time execution.

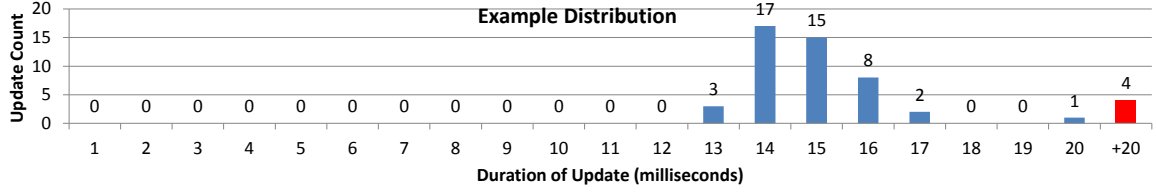


Figure 19. Sample Performance Graph. This update duration frequency distribution graph shows the wait times of a 1 second simulation with a frame rate of 50 Hz. The majority of updates lasted 14 to 16 ms, indicating real-time execution, but also high processor utilization (distributions closer to 20 ms indicate high utilization while distributions closer to 0 ms indicate low utilization and thus more room for the simulation of additional swarming UAVs).

Simulations demonstrated in this thesis are assessed for real-time execution by recording the duration of each simulation tree update during the first 10 minutes of each simulation run. Update durations are rounded up to the nearest integer value in milliseconds (e.g. 2.0324 ms is rounded up to 3 ms) and plotted as a frequency distribution graph. The shape and position of simulation update duration distributions should provide strong indications of simulation performance. For example, Figure 19 depicts a sample distribution of a simulation with a 50 Hz frame rate and total duration of approximately 1 second. The corresponding duration for each of the 50 simulation tree updates show that the majority of updates lasted between 14 and 16 ms. One of the updates lasted the entire refresh period (i.e. 20 ms) while four updates lasted longer than the refresh period indicating real-time simulation was not achieved during those update periods.

A 50 Hz frame rate (i.e. 20 ms frames) is used to satisfy requirements of the hardware device used in this thesis—the Pixhawk autopilot requires a minimum 50 Hz stream of measurement data during HIL simulation. Thus, real-time execution is considered achieved during updates with a duration less than or equal to 20 ms. Furthermore, because some updates lasting longer than 20 ms are recoverable (e.g. the second update for Process B in Figure 18), the ratio of real-time updates will be monitored. Specifically, percent of real-time execution, P_{rt} , is calculated as

$$P_{rt} = \frac{n_{rt}}{n_{total}} \quad (1)$$

where n_{rt} denotes the number of updates that lasted 20 ms or less, and n_{total} is the number of updates evaluated. In this way, the P_{rt} for the example in Figure 19 is $\frac{46}{50}$, or 92.0%. For this work, $P_{rt} \geq 90\%$ is considered optimal, $90\% > P_{rt} \geq 70\%$ is considered acceptable, and $P_{rt} < 70\%$ is unacceptable.

Max Swarm Size. Framework scalability is ultimately assessed in this thesis by determining the maximum swarm size while maintaining real-time execution in accordance with the P_{rt} metric above. Additionally, scalability of purely simulated swarms (i.e. using only simulated autopilots) is compared to that of swarms fully configured for HIL (i.e. using only Pixhawk autopilots). Such comparison will indicate the difference in resource requirements between simulated autopilots and the processing of hardware interface data streams.

3.4.4 Swarm Development Scenario.

The following scenario is used to demonstrate the capabilities and limitations of the swarm simulation framework: Imagine you are a swarm developer and wish to understand the behavior of a swarm of UAVs programmed with Reynolds flocking rules. Specifically, you want to understand how such a swarm will interact with three piloted aircraft. Will they merge into a tight formation and then drift off in a random direction? Will they split into smaller formations, each following one of the piloted UAVs? Or will they crash into each other and fall to the ground?

The next chapter walks through the steps a developer may take when applying the proposed swarm simulation framework to answering these questions. Four demonstrations are presented, each of which builds off the previous demonstration to show capabilities and limitations of the framework. The first demonstration introduces the

swarm algorithm and shows how to implement it in the OCA of a single UAV. The next demonstration expands upon the first by applying the swarm algorithm to multiple UAVs, increasing the swarm size until an upper limit is reached (i.e. until the processor cannot update the entire simulation tree within the 20 ms refresh period, assuming a 50 Hz real-time refresh rate). The third demonstration incorporates a single Pixhawk autopilot for HIL simulation. Finally, the fourth demonstration expands HIL simulation to multiple devices, increasing the device count until an upper limit is reached.

The scenario described here will constitute the domain used in the experiment presented in the next chapter. Three navigating (non-swarming) UAVs serve as “piloted” aircraft and continuously fly three distinct patterns through 10 common waypoints. The waypoints are distributed within a 4.5 by 3.4 nautical mile rectangular grid at various altitudes ranging from 13,700 to 14,300 feet MSL (Figure 20). The navigating UAVs are designated as N1, N2, and N3 which fly the routes shown in Figure 21. Swarming UAVs programmed with Reynolds flocking rules are added to the environment and monitored.

Framework demonstrations use the above scenario to demonstrate the OCA swarm development sandbox in which developers can apply and test swarm algorithms in customized configurations with high confidence that resulting swarm behaviors will translate to real world flight, thus validating or disproving the swarm control strategy under development without risking physical aircraft. All referenced code can be found at <https://github.com/derekworth/SwarmSim>.

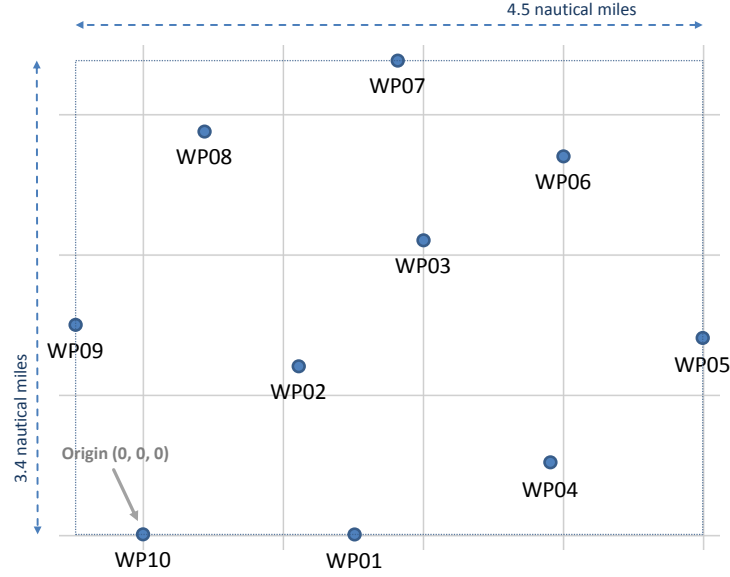


Figure 20. Scenario Common Waypoint Field. Each simulation contains 10 common waypoints that are distributed within a 20 square mile region with the origin directly below WP10, at approximately 39.0084648°N, -104.8887177°W, and 0 feet MSL.

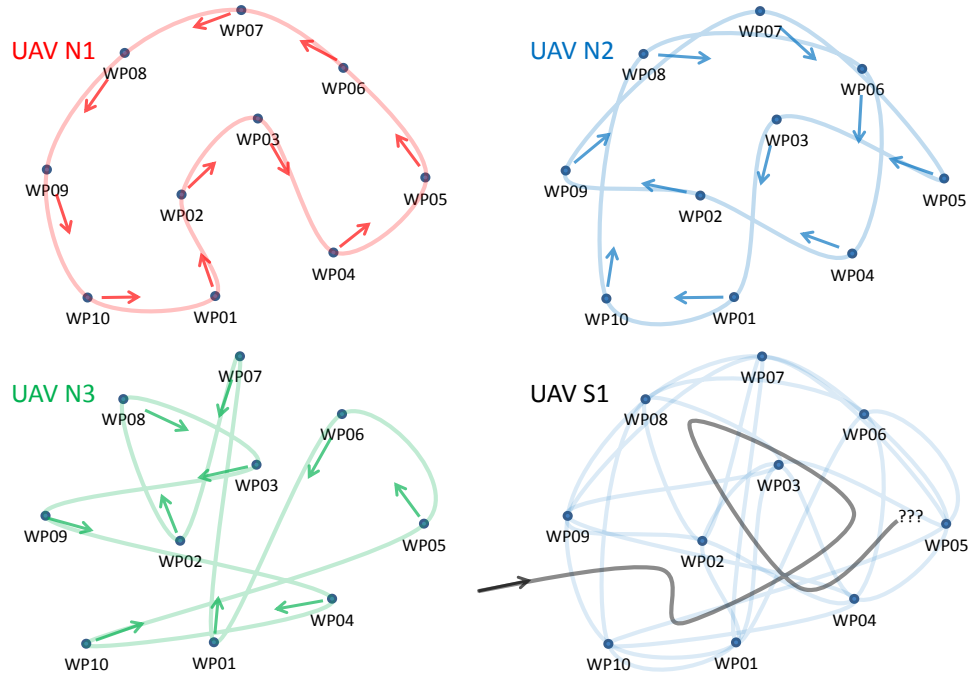


Figure 21. UAV Paths. UAVs in navigation mode repeat their respective pre-programmed paths through the 10 common waypoints while swarming UAVs determine their path using Reynolds Flocking rules and information gathered about neighboring UAVs.

Flight Dynamics Model. As previously mentioned, the FDM used for this proof of concept is JSBSim with an XML-defined *Sig Rascal 110 ARF* aircraft model (Figure 22)—which “is based on the Rascal110 from FlightGear, and adapted for use in the ArduPilot test system” [18]. Although JSBSim accommodates virtually any aircraft model, the Sig Rascal was chosen because it is an archetypal air frame with widespread use and development in the hobbyist and drone communities. It falls in the *small fixed-wing UAV* class with a single propeller, traditional controls configuration (ailerons, rudder, elevator), and wingspan of 9.17 feet.



Figure 22. Sig Rascal 110 ARF Model.

SIMDIS. To view the dynamic interactions of the swarm behaviors under development, Distributed Interactive Simulation (DIS) is enabled using the OpenEagles network class, *DisNetIO*, and configured to broadcast DIS packets over the localhost loopback network interface. Such packets are intercepted and translated by a set of software tools known as SIMDISTM, which provide 3D interactive graphical and video display of simulation data produced by the simulation demonstrations. Figure 23 shows the primary tools pertinent to the scenario described above. SIMDIS serves as the visual interface for all demos in this thesis. For visualization purposes only, UAVs rendered in SIMDIS appear as MQ-1 Predators due to a limited availability of icon models. This detail has no impact on the UAV flight characteristics, which are defined by the Sig Rascal 110 FDM (i.e. simulations consist of UAVs that fly like Sig Rascals, but look like Predators).

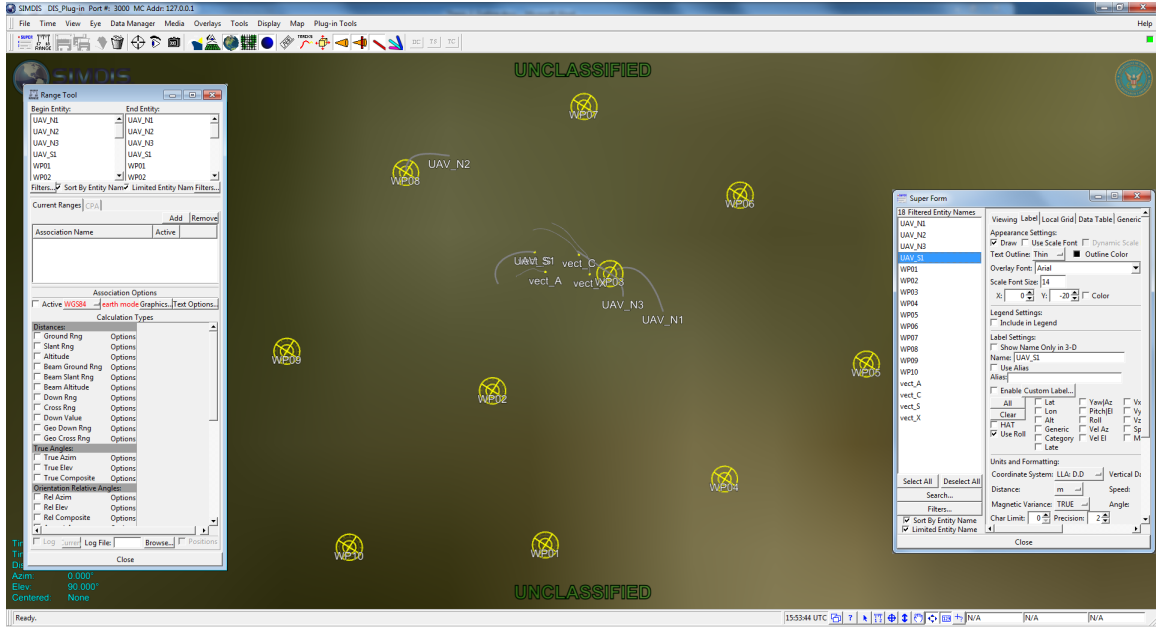


Figure 23. SIMDIS Overview. SIMDIS intercepts and interprets the DIS packets broadcasted by the simulation. The Range Tool on the left provides ranging information (i.e. difference in altitude, slant range, etc.) between two or more entities while the Super Form on the right provides view customization (i.e. adding/removing labels, data tables, local grids, etc.) and the ability to focus the view on a specific entity. The main window provides point-and-click interactivity, allowing users to pan, tilt, and zoom freely while the simulation is running.

3.5 Summary

This chapter showed how OpenEagles can be extended to simulate swarms of UAVs using hardware-in-the-loop and described a swarm development scenario that will be used in the next chapter to assess the limitations and capabilities of the proposed framework.

IV. Design Demonstrations/Results

This chapter presents four demonstrations to highlight the capabilities and limitations of the proposed swarm simulation framework. Each demonstration consists of a setup followed by observations made during and after implementation. Both qualitative (observations made on framework reusability, usability, and modularity) and quantitative (performance measurements assessing scalability) analysis are conducted for each demonstration.

4.1 Applying Reynolds Flocking Rules

In this demonstration, simple Reynolds flocking rules (separation, alignment, and cohesion) as detailed by Brundage [6] are introduced and implemented in the onboard control agent (OCA) to provide a basic swarm behavior and demonstrate swarm development encapsulation within the OCA. A single swarming UAV with visible Reynolds vectors are implemented as well as three non-swarming UAVs.

4.1.1 Setup.

Autopilot. Only simulated autopilots are used in this demonstration to test Reynolds-based swarm behaviors. The simulated autopilot class (*SimAP*), which extends the *SwarmAutopilot* class for dynamic waypoint following (DWF) functionality, utilizes proportional, integral, and derivative (PID) gain control logic to generate commanded roll, pitch, and yaw outputs with constant maximum throttle. Specifically, the simulated autopilot steers the aircraft in the heading of the next waypoint using roll and yaw controls while waypoint altitude is achieved using pitch controls. Gains are adjustable and specified in Eaagles Definition Language (EDL) files and tuned for the Sig Rascal 110 model for stable flight and navigation. The default OpenEaagles

Navigation class is used to provide heading information and course correction. The navigation system of the “piloted” (non-swarving) UAVs are programmed with the 10 common waypoints and continuously cycle through them in their corresponding pattern order.

Calculating Reynolds Vectors. Swarm algorithms (i.e. Reynolds rules) reside in the OCA. *Alignment* and *cohesion* vectors steer swarming UAVs together while *separation* vectors prevent them from colliding into each other (Figure 24). The OCA calculates alignment and cohesion vectors (*vect_A* and *vect_C*) by averaging the velocity and position vectors respectively of surrounding UAVs, while calculating the separation vector (*vect_S*) by averaging the negation of all surrounding UAV position vectors within a desired separation distance as follows, where \vec{p} represents a position vector, \vec{p}_i the vector pointing from the origin UAV to a neighboring UAV, n the quantity of neighboring UAVs within the pre-defined range, and DS the desired separation which is set to 1000 meters for all swarming UAVs:

$$\vec{p}_i = \vec{p}_{uav} - \vec{p}_{neighbor-i} \quad (2)$$

$$vect_S = \frac{\sum_{i=1}^n \vec{p}_i \times \left\{ \frac{DS}{\|\vec{p}_i\|} \right\}^2}{n} \quad (3)$$

The dynamic waypoint vector (*vect_X*) is calculated by summing the three Reynolds vectors after multiplying them by corresponding scale factors. Scale factors used here are 0.5, 10.0, and 1.0 for the separation, alignment, and cohesion vectors respectively. Finally, the dynamic waypoint vector is directly translated into a waypoint (i.e. latitude, longitude, and altitude offsets from the current position of the swarming UAV) and passed to the autopilot for autonomous navigation. For visualization of the vectors, four “dummy” Player entities are added to the simulation tree whose positions

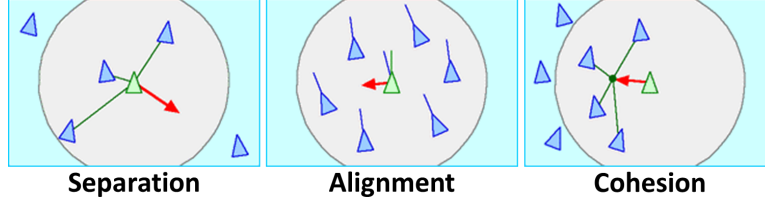


Figure 24. Reynolds Flocking Rules. *Separation* steers to avoid crowding local flockmates, *alignment* steers toward the average heading of local flockmates, and *cohesion* steers to move toward the average position of local flockmates. (Originally presented in [38] and reproduced as an element of the public domain.)

are updated by the OCA and correspond to the four vectors. The OCA recalculates the vectors once every simulation tree update resulting in a smooth and continuous visualization of the Reynolds flocking rules.

4.1.2 Observations.

Accuracy of Simulation. During the design and tuning of the autopilot, qualitative analysis showed accuracy and responsiveness of flight control inputs to the Sig Rascal FDM. Preset flight control inputs were configured to induce specific flight maneuvers. The simulated UAV performed as expected and thus increased confidence in the accuracy of the FDM. Figure 25 shows some of the resulting maneuvers performed.

Quality Attributes. After establishing the *UAV*, *OnboardControlAgent*, and *SwarmAutopilot* classes, the framework provided strong usability. *SimAP* (extending *SwarmAutopilot*) contained three additional methods corresponding to the aircraft roll, pitch, and yaw controls (throttle remained constant at 100%). The *OnboardControlAgent* interacted with the *SimAP* by calling upon the *setWaypoint* method inherited from *SwarmAutopilot* and also contained three methods, corresponding to the three Reynolds vectors which easily encapsulated the development and final implementation of the swarm algorithm. Integrating a single implementation of the

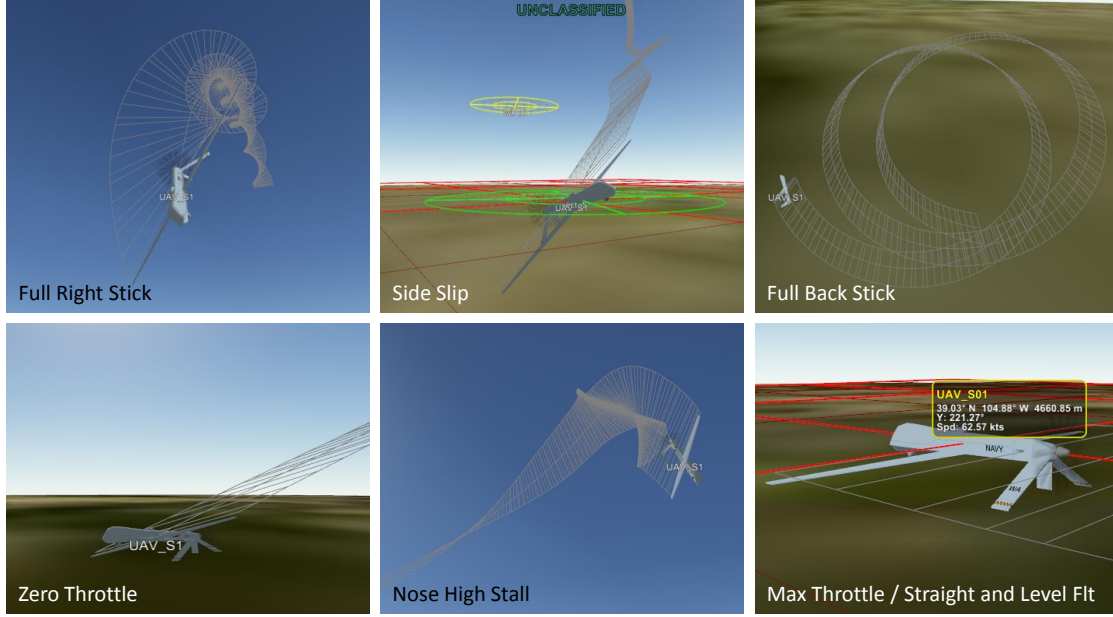


Figure 25. Flight Maneuvers. Several different flight maneuvers were performed to test yaw, pitch, roll, and throttle control accuracy and responsiveness of the Sig Rascal 110 FDM.

autopilot class (i.e. SimAP) into both the navigating and swarming UAVs without modification demonstrates code reusability.

Swarm Behavior. Simulation shows visual representation of Reynolds vectors and corresponding dynamic waypoint. Figure 26 highlights each vector (indicated with black arrows). The separation vector points opposite to UAV N2—the only UAV withing desired separation range. The alignment vector points to the average direction of travel for the three navigating UAVs. The cohesion vector points to the average position of the three navigating UAVs. Lastly, the dynamic waypoint vector points to the sum of the three scaled Reynolds vectors. The vectors update every simulation refresh period and therefore provide a smooth and detailed outline of vector history as indicated by the simulated residual trail following each vector. This unnecessarily high OCA refresh rate is for demonstration purposes only and a lower refresh rate is used in subsequent demonstrations. As anticipated, the swarming

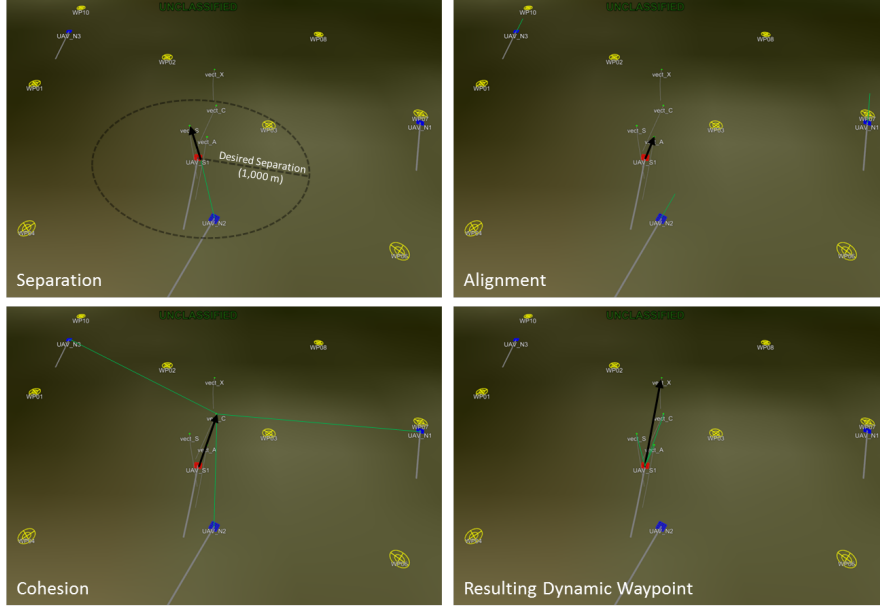


Figure 26. Simulated Reynolds Vectors. The separation vector (top left) shows the combined repulsive “force” of any surrounding UAVs within its desired separation range—only UAV N2. The alignment vector (top right) shows the average velocity vectors of surrounding UAVs. The cohesion vector (bottom left) points to the average position of surrounding UAVs. The dynamic waypoint vector (bottom right) is derived from summing these three vectors together.

UAV continuously “chased after” the dynamic waypoint, vect_X, demonstrating the ability of the OCA to direct the autopilot.

Performance Analysis. During a 10-minute simulation of the single swarming UAV interacting with the three navigating UAVs, all 30,000 updates (i.e. one update every 20 ms) were recorded. Figure 27 shows the frequency distribution of simulation update durations, wherein the P_{rt} is 100% and the majority of updates lasted approximately 4 ms. Such results indicate low resource utilization and demonstrate an ability to accommodate additional swarming UAVs.

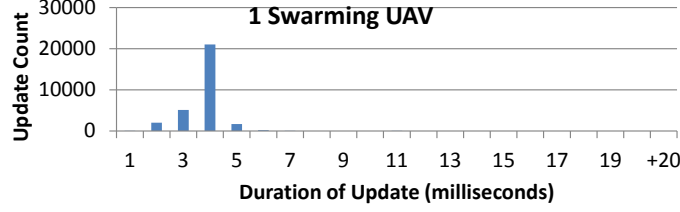


Figure 27. Performance Graph - Single Swarming UAV. This graph shows the update durations recorded after each simulation tree update during a 10-minute simulation of a single swarming UAV and three navigating UAVs (simulated autopilots only).

4.2 UAV Swarms with Simulated Autopilots

This demonstration extends the previous one by inserting additional swarming UAVs to determine the maximum swarm size (with simulated autopilots) capable of executing in real-time.

4.2.1 Setup.

UAV subsystems (OnboardControlAgent and SimAP) and the “piloted” UAV configuration (three UAVs navigating through 10 common waypoints) used in the previous demonstration are included here. However, the four “dummy” player entities (visually representing dynamic waypoint and Reynolds vectors) are removed. Swarming UAVs are randomly positioned within two nautical miles of the scenario origin. The quantity of swarming UAVs is increased until an upper limit is reached (i.e. the maximum swarm size that still executes in real-time). Anticipating a polynomial growth, $O(n^2)$, in peer-to-peer velocity and position vector queries for dynamic waypoint calculations, the OCA refresh rate is reduced from the unnecessarily high frequency of 50 Hz to 0.2 Hz to more accurately measure the impact of simulating additional UAVs.

4.2.2 Observations.

Increasing swarm size progressively lengthened update durations resulting in an eventual lag behind real-time execution. Table 3 lists the percent of real-time execution (P_{rt}) values for 10-minute simulations of various swarm sizes while Figure 28 shows their performances, measured in recorded update durations. The observed maximum swarm size capable of an acceptable P_{rt} value was 64 swarming UAVs (Figure 29). Although real-time execution limits the swarm size, larger swarms can be accurately simulated outside real-time. Figure 30 shows the behavior of a much larger swarm which executed about five times slower than real-time.

Table 3. Percent Real-time Execution

Swarm Size	P_{rt}
1	100.0%
10	100.0%
20	100.0%
30	100.0%
40	100.0%
50	99.9%
60	97.3%
61	91.5%
62	78.5%
63	74.1%
64	74.4%
65	67.3%
66	52.0%
67	30.4%
70	21.8%

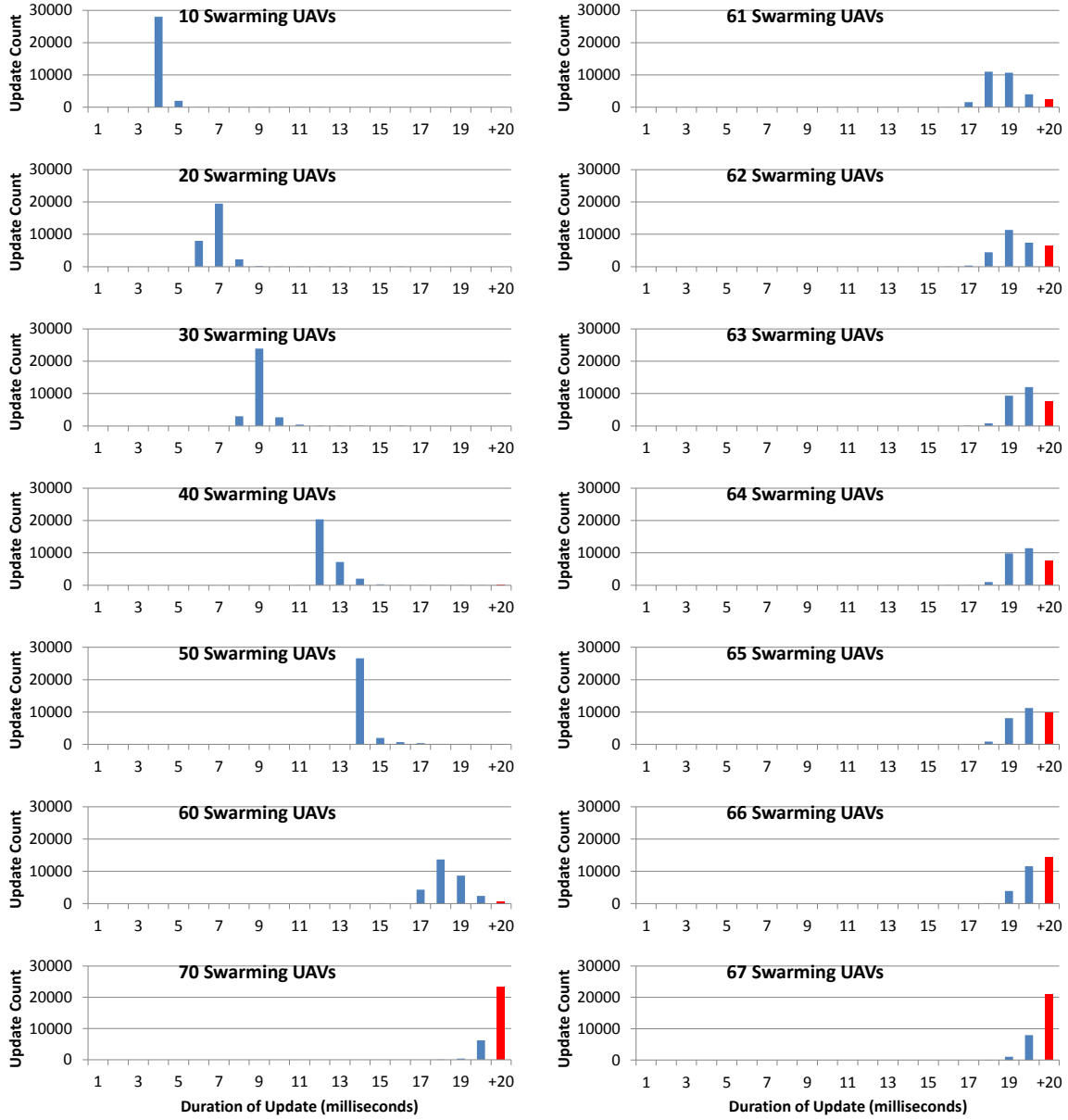


Figure 28. Swarm Size Performance Comparison. Each of the above simulations lasted for a duration of 10 minutes. During execution, the duration of each simulation tree update was recorded. Blue bars reflect updates finishing within their 20 ms frame (i.e. real-time execution achieved) while red bars indicate the number of updates extending past the 20 ms frame (i.e. updates lagging behind real-time).

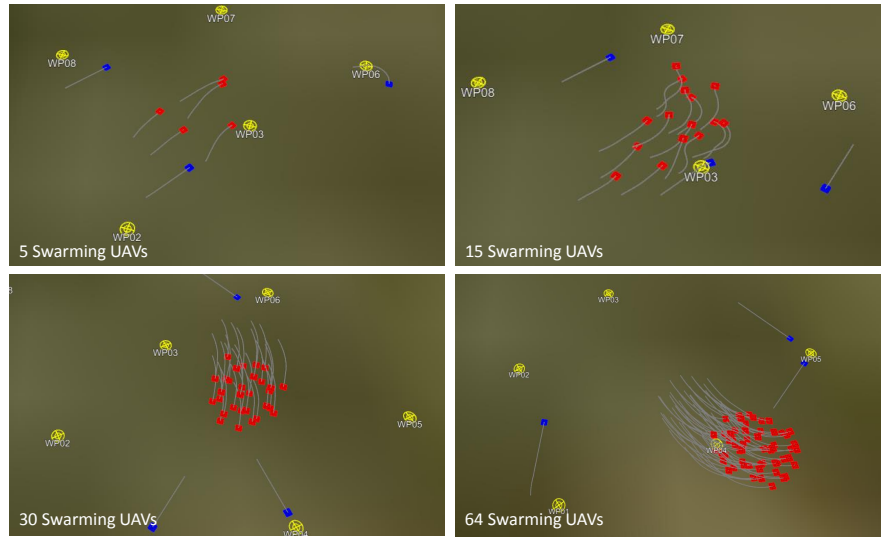


Figure 29. Real-time Swarms. This figure shows the gradual increase in swarm size until an upper limit capable of executing in real-time was achieved. Experimentation resulted in a maximum size of 64 swarming UAVs (depicted in the bottom right).

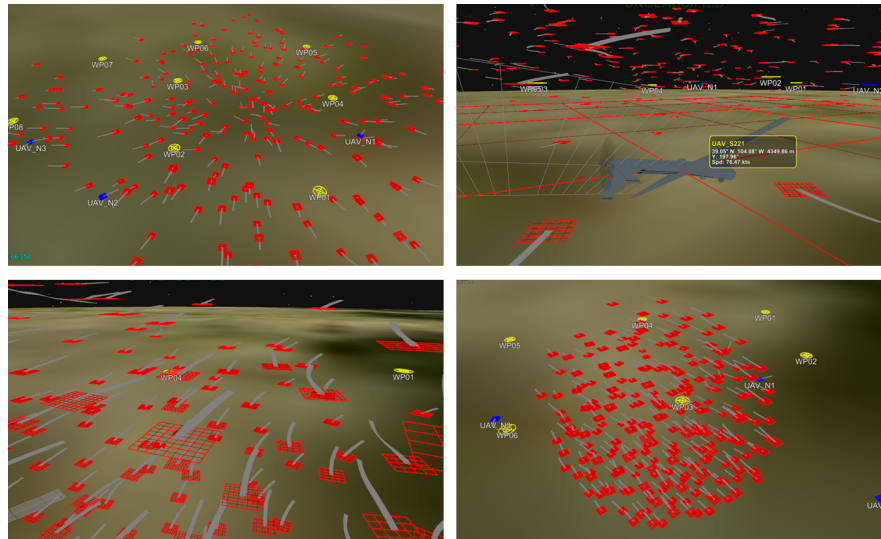


Figure 30. Large Swarm. Shown here are multiple views of 240 swarming UAVs. After randomly positioning the swarming UAVs within two nautical miles of the common waypoint field, they merge together (top left) to eventually form a tight spherical formation (bottom right) that loiters near the three “piloted” UAVs without any collisions.

4.3 Applying Hardware-in-the-Loop Simulation

This demonstration transitions the previous demonstrations from swarms using purely simulated autopilots to simulations with hardware autopilot integration (i.e. hardware-in-the-loop). Specifically, a hardware interface to the Pixhawk autopilot platform, designated *PixhawkAP*, was implemented such that it passes simulated UAV telemetry information from the FDM to the Pixhawk in exchange for flight control signals returned to the FDM. Additionally, it sends dynamic waypoint updates from the OCA to the Pixhawk. *SimAP* for the swarming UAV was replaced with *PixhawkAP*. The configuration for the navigating UAVs remained unchanged. Finally, the dynamic waypoint and Reynolds vector visuals are enabled for the swarming UAV.

4.3.1 Setup.

Preparing the Pixhawk for HIL. Before incorporating the Pixhawk for HIL simulation, the ‘px4fmu-v2.default.px4’ flight stack was installed using QGroundControl v2.3.0. QGroundControl also served as the HIL interface between FlightGear v3.4.0 (running the same Sig Rascal 110 FDM) and a Pixhawk autopilot during parameter tuning. To increase flight stability while swarming, the FW_P_LIM_MAX parameter (positive pitch limit) was changed from its default value of 45.0 to 15.0. FW_P_RMAX_POS and FW_P_RMAX_NEG (maximum positive and negative pitch rates) were both changed from their default value of 60.0 to 10.0. All other parameters remained unchanged from their default values. The Pixhawk is armed and set to *Auto* mode prior to simulated flight using QGroundControl which sets its base mode set to 189 (corresponding to HIL, stabilized, guided, auto, and custom modes enabled; safety armed; testing and manual input disabled). Additionally, the Pixhawk safety

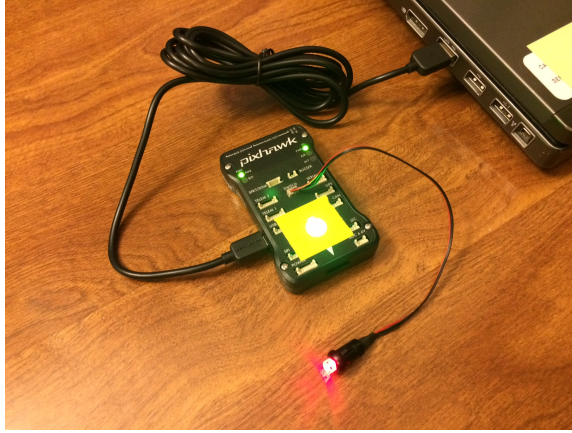


Figure 31. Pixhawk Hardware Setup. A single Pixhawk autopilot is connected to the simulation over USB (COM Port 5). The safety switch (i.e. button at the end of the extending wire) illuminates solid red when safety is disabled. The multi-color LED flight status indicator (covered with yellow tape in the image) illuminates solid green indicating system armed, GPS lock acquired, and ready to fly. Additionally, it flashes green when receiving waypoints. Any other light pattern during simulation indicates an error condition.

switch must be installed and toggled to illuminate solid red. The final setup is shown in Figure 31.

To test the Pixhawk’s ability to accommodate DWF, a single waypoint was programmed into the Pixhawk and periodically updated to various locations mid-flight (Figure 32). The Pixhawk followed the waypoint as it dynamically changed (i.e. performed DWF) as anticipated, confirming its utility in the swarm scenario. For the purpose of this and the next demonstration, all HIL mode initialization for the Pixhawk is accomplished through QGroundControl paired with an instance of FlightGear. Once HIL simulation with DWF is established, the Pixhawk is disconnected from QGroundControl and connected to the swarm simulation framework via a PixhawkAP hardware interface.

Hardware Requirements. HIL integration of the Pixhawk autopilot requires a serial connection over USB, MAVLink communications (sending and receiving streams), and translation of telemetry and waypoint data flowing between the Pixhawk autopilot and simulation (i.e. FDM and OCA). A serial connection to the

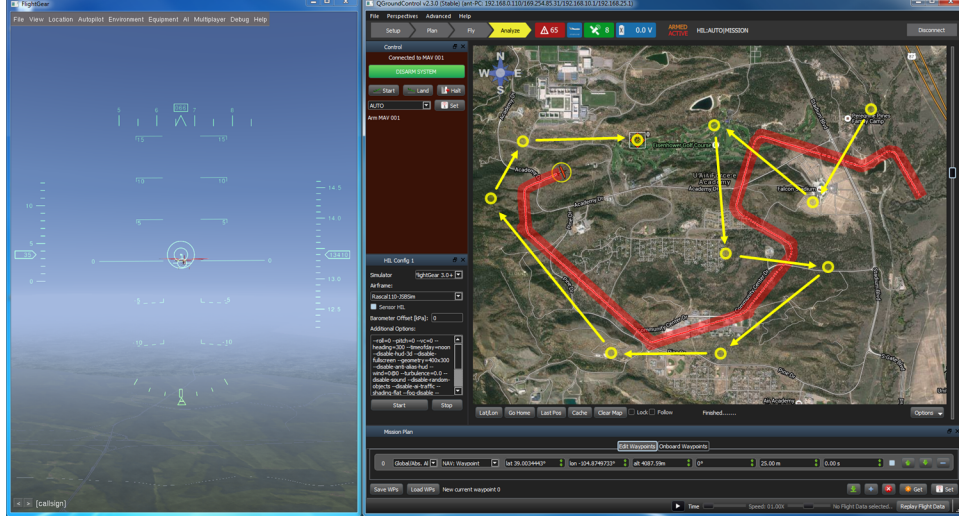


Figure 32. Dynamic Waypoint Following with QGroundControl. QGroundControl acts as the hardware interface between simulation (i.e. FlightGear pictured on the left) and a Pixhawk autopilot. Yellow circles indicate dynamics waypoint updates while yellow arrows (starting from the top-right) show the order in which the updates took place. The line highlighted in red indicates the actual UAV flight path.

Pixhawk was established using the CSerial application program interface (API) [2]. A C++ version of the MAVLink common message library [31] provides the capability to encode/decode messages to and from the Pixhawk. The MAVLink common message set found at <https://pixhawk.ethz.ch/mavlink/> details the MAVLink message requirements used by the Pixhawk and thus guided the code development of data translation logic implemented in the PixhawkAP class.

Timing. Table 4 summarizes necessary MAVLink message traffic (from the perspective of the simulation) during HIL simulation along with message rates and sizes. To enable HIL mode, the Pixhawk requires a steady supply of valid measurement data packed into HIL_SENSOR and HIL_GPS MAVLink messages, streamed in at 50 Hz and 10 Hz respectively. Additionally, the Pixhawk requires HEARTBEAT messages streamed in at 1 Hz. To accommodate each corresponding message transmission rate in real-time, the PixhawkAP synchronizes each rate with the computer’s system clock.

Table 4. HIL MAVLink Message Traffic

Msg ID	Message	Direction	Size (Bytes)	Rate
0	HEARTBEAT	both	17	1 Hz
39	MISSION_ITEM	send	45	as needed
40	MISSION_REQUEST	receive	12	as needed
44	MISSION_COUNT	send	12	as needed
46	MISSION_ITEM_REACHED	receive	10	as needed
47	MISSION_ACK	receive	11	as needed
91	HIL_CONTROLS	receive	50	37 Hz
107	HIL_SENSOR	send	72	50 Hz
113	HIL_GPS	send	44	10 Hz
253	STATUSTEXT	receive	59	as needed

Data Buffers. Because the CSerial API does not accommodate interrupts, polling is used to read data from the serial buffer in 128 byte blocks. Specifically, during each simulation frame, the PixhawkAP interface queries the number of bytes waiting in the serial buffer. If more than 128 bytes are waiting, the PixhawkAP enters a loop and continuously pulls data from the buffer, 128 bytes at a time, until the buffer contains less than 128 bytes—only then does the PixhawkAP break out of the loop and continue processing the remaining time-critical tasks for that frame. As long as the PixhawkAP interface reads data from the buffer faster than the Pixhawk can write to it, no bytes are dropped. Lastly, waypoints are programmed onto the Pixhawk using the waypoint protocol (detailed at http://qgroundcontrol.org/mavlink/waypoint_protocol).

4.3.2 Observations.

Quality Attributes. Use of the Pixhawk autopilot for DWF requires no familiarity with its code base. Instead, it requires an understanding of how it behaves. In other words, in order to use the Pixhawk, swarm developers must understand “what” it does and not “how” it does it. Such encapsulation of implementation details greatly simplifies its integration into HIL simulation while promoting usability and modularity. In this case, DWF requires the exchange of waypoints (generated

by the OCA—this is where swarm development takes place) and measurement data (generated by the FDM) for control signals (generated by the Pixhawk). For example, the Pixhawk sends HIL_CONTROLS messages containing roll, pitch, yaw, and throttle controls only after receiving HIL_SENSOR and HIL_GPS messages (while in HIL mode) containing acceleration, gyroscope, magnetometer, pressure, temperature, position, velocity, and course over ground measurements. Subsequently, it autonomously drives the simulated air vehicle toward waypoints (latitude, longitude, and altitude) specified by MISSION_ITEM messages.

However, such interaction with the Pixhawk does require intimate knowledge of how it communicates. The PixhawkAP interface must accurately read and translate measurement data from the FDM and waypoints from the OCA into MAVLink messages the Pixhawk can understand. Otherwise, the Pixhawk will respond with anomalous control signals or no signals at all. To accomplish accurate data translations between the simulation and the Pixhawk, detailed API documentation is essential. Limited documentation results in loss in development productivity. Fortunately, the MAVLink common message set provides the translation details necessary to communicate accurate simulation state information to the Pixhawk while accepting control signal in return.

Performance Analysis. Analysis of update durations shows the HIL simulation maintained real-time execution (Figure 33). However, simulations with a single swarming UAV driven by a Pixhawk autopilot averaged update durations similar to that of simulations with 38 swarming UAVs driven by simulated autopilots, indicating a substantial reduction in scalability when transitioning to HIL simulation—which is caused mostly by the high rate of simulation-to-MAVLink data translations when generating and sending HIL_SENSOR and HIL_GPS messages.

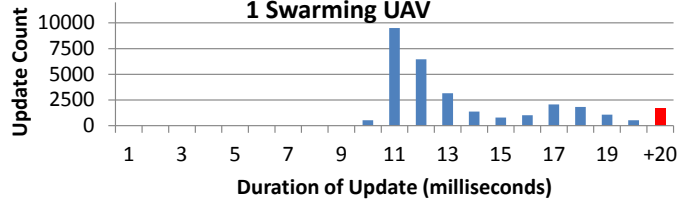


Figure 33. Performance Graph - Single Swarming UAV (HIL). This graph shows the update durations of a 10-minute simulation consisting of one swarming UAV integrated with a Pixhawk autopilot (for HIL simulation) and three navigating UAVs with simulated autopilots. The smaller of the two local peaks (at approximately 17 ms) indicates periodic increases in resource utilization outside the main refresh period—namely the generation and transmission of 10 Hz HIL_GPS MAVLink messages.

To provide control signals (i.e. HIL_CONTROLS) to the simulation, the Pixhawk requires accurate measurement data streamed in at high rates (50 Hz for HIL_SENSOR and 10 Hz for HIL_GPS messages). The generation of each HIL_SENSOR message requires many costly multiplication operations associated with magnetometer calculations as well as pressure and temperature conversions. Timestamps taken before and after the *sendHilSensor* method call indicated the translation process—which included pulling measurement data from the FDM, converting it to a HIL_SENSOR MAVLink message, and sending the message over the serial link to the Pixhawk—took approximately 11 ms to complete. Similarly, HIL_GPS messages require many multiplication operations and external method calls. Timestamps taken before and after the *sendHilGps* method call indicated the translation process took approximately 7 ms to generate and send each HIL_GPS message.

Figure 33 shows two peaks—a large peak at 11 ms and a smaller one at approximately 17 ms. The decrease in average wait times from using a simulated autopilot to incorporating a Pixhawk in HIL simulation (indicated by the larger peak) is due to the HIL_SENSOR messages generated every refresh period while the smaller peak results from a periodic (10 Hz) increase in resource utilization that occurs outside the 50 Hz frame rate. In other words, most updates lasted approximately 11 to 13 ms because a

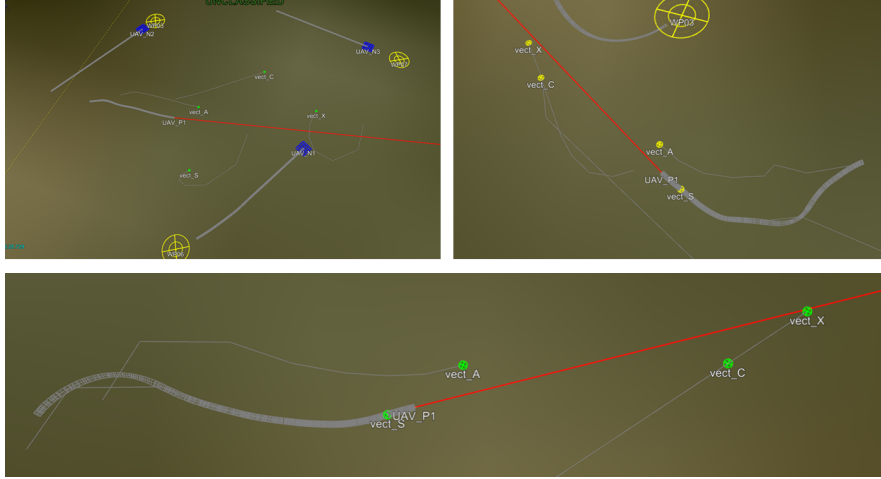


Figure 34. Reynolds Vectors with HIL Simulation. The Pixhawk autopilot provides flight control signals to swarming UAV P1’s FDM, ultimately driving it to continuously follow the dynamic waypoint (which the OCA updates every five seconds). The red line indicates P1’s instantaneous direction of travel.

HIL_SENSOR message is generated and sent every frame (which takes approximately 11 ms), while the majority of remaining updates last 17 to 19 ms because every five frames, a HIL_GPS message is generated and sent (which takes approximately 7 ms) in addition to the HIL_SENSOR message.

Behavior. Because *PixhawkAP* extends *SwarmAutopilot*, PixhawkAP seamlessly replaces SimAP in the swarming UAV (re-designated as P1). When the OCA sets the dynamic waypoint using the *setWaypoint* method, PixhawkAP forwards that waypoint over USB to the Pixhawk using the waypoint protocol. The exchange of measurement data for control signals takes place as mentioned above. Figure 34 shows the resulting behavior of incorporating the Pixhawk autopilot in HIL simulation using the swarm simulation framework—P1 behaves similar to S1 from the first demonstration, chasing *vect_X*, thus confirming a successful hardware-in-the-loop implementation that is directed by the onboard control agent.

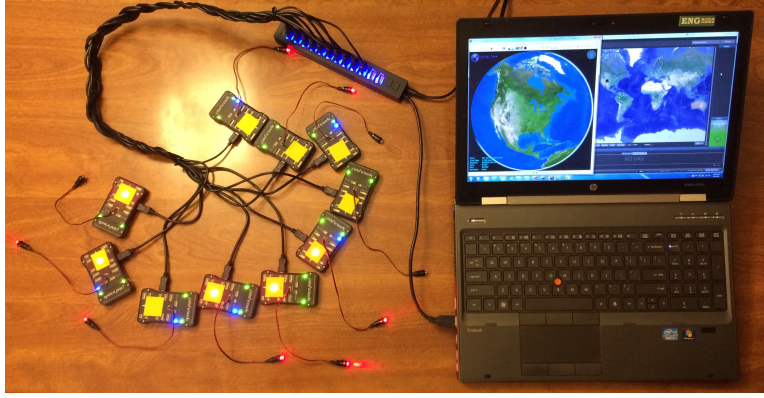


Figure 35. Multi-Pixhawk Hardware Setup. The laptop shown here (right) is the platform used throughout this thesis. Each Pixhawk (left) is connected to a USB 2.0 hub (top center) which is connected to a USB 2.0 port on the side of the laptop.

4.4 UAV Swarm with Hardware Autopilots

This final demonstration expands HIL simulation to multiple swarming UAVs integrated with Pixhawk autopilots. Additional swarming UAVs are added to the simulation until a maximum swarm size is reached (i.e. largest swarm size with an acceptable P_{rt} value).

4.4.1 Setup.

The setup from the single HIL demonstration is used here with the dynamic waypoint and Reynolds vector visuals removed. Parameters from the previous demonstration are copied to additional Pixhawk autopilots. HIL mode is initialized separately for each individual Pixhawk using QGroundControl. Each Pixhawk is connected over USB 2.0 (Figure 35) and automatically assigned a virtual COM port. COM port associations between the PixhawkAP interfaces and corresponding Pixhawk autopilots are defined in an EDL file. Similar to the second demonstration, swarming UAVs are randomly positioned within two nautical miles of the scenario origin.

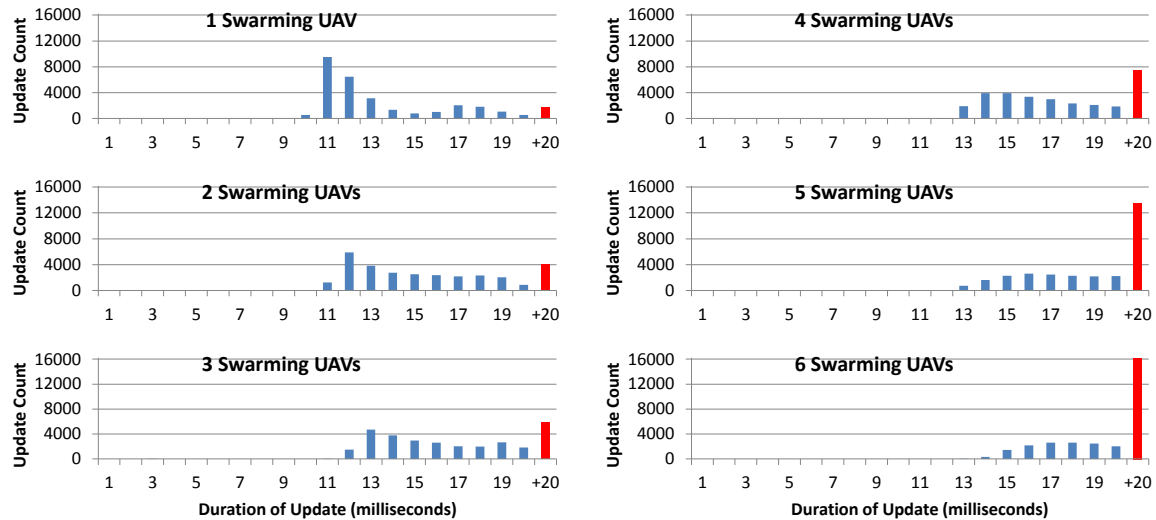


Figure 36. Swarm Size Performance Comparison (HIL). The graphs above show a linear degradation of simulation performance as more Pixhawk autopilots are added.

4.4.2 Observations.

Performance Analysis. The update duration distributions for up to six swarming UAVs in HIL simulation are shown in Figure 36 with corresponding P_{rt} values listed in Table 5. Acceptable P_{rt} values were observed in swarms with up to four UAVs simultaneously interfacing with Pixhawk autopilots in HIL simulation.

Table 5. Percent Real-time Execution (HIL)

Swarm Size	P_{rt}
1	94.3%
2	86.7%
3	80.5%
4	74.9%
5	54.8%
6	46.1%

Behavior. The various swarm sizes demonstrated similar behavior to that of the same size using only simulated autopilots. The UAVs merged together into a tight formation and loitered within the common waypoint field, near the three navigating UAVs. Figure 37 shows the coordinated behaviors for different swarm sizes. Although a maximum size of only four swarming UAVs could execute in HIL simulation with an

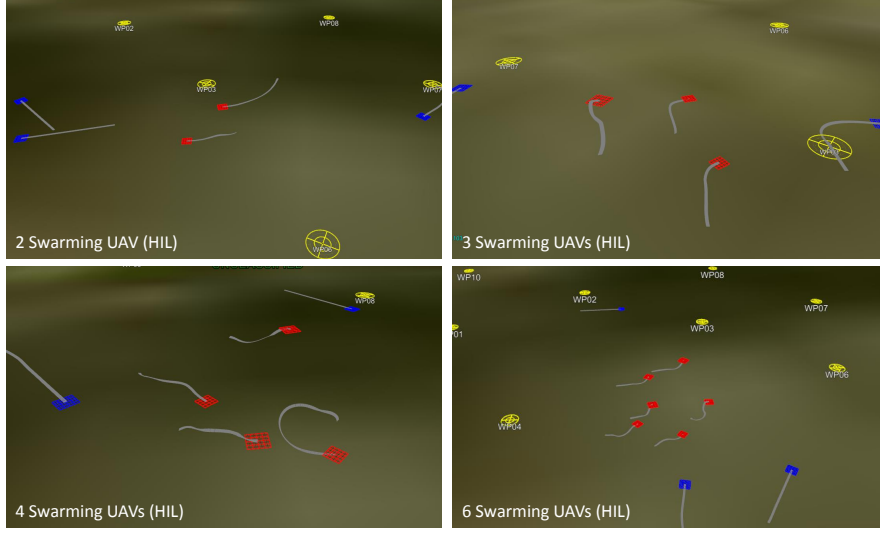


Figure 37. Swarming During HIL Simulation. The swarming UAVs (highlighted in red) are driven by Pixhawk autopilots during HIL simulation and behave similar to swarming UAVs driven by simulated autopilots from the second demonstration.

acceptable P_{rt} value, swarms of size five and six also demonstrated stable Reynolds flocking behavior.

4.5 Summary

The four demonstrations explored in this chapter stepped through various stages of swarm behavior development and modeling using the proposed swarm simulation framework. Simple Reynolds flocking rules served as the swarm algorithm while their implementation showcased how OCA encapsulation paired with DWF promotes and facilitates swarm behavior development. The ability to add, remove, and interchange different components (e.g. swapping SimAP with PixhawkAP) without adversely impacting other components or interactions within the framework demonstrates strong modularity. Additionally, the framework successfully accommodated 61 swarming UAVs configured with purely simulated autopilots and four swarming UAVs integrated with Pixhawk hardware autopilots, all while remaining synchronized with real-time.

V. Conclusions and Future Work

This thesis presented an extension to the OpenEagles simulation framework that enabled the testing and development of scalable, modular, and realistic UAV swarms in simulation using hardware-in-the-loop (HIL). The OpenEagles simulation tree hierarchy promotes high modularity and scalability for swarm component integration. As demonstrated in Chapter IV, the use of onboard control agents (OCA) paired with dynamic waypoint following (DWF) facilitates swarm development encapsulation, ultimately providing a “sandbox” wherein developers can implement and test swarm algorithms and control strategies before applying them to real aircraft. Furthermore, JSBSim—a trusted flight dynamics model (FDM) already integrated into OpenEagles—provides realistic aircraft modeling for high confidence that simulated swarm or UAV behaviors will transition to real flight tests.

However, when incorporating hardware devices for HIL simulation, the maximum swarm size was only four UAVs due to the amount of processing required to translate data between hardware and the simulation—namely generating and sending HIL_SENSOR and HIL_GPS MAVLink messages. The maximum swarm size was much larger when utilizing only simulated autopilots, which shows the framework is capable of high scalability. Furthermore, the framework is capable of accommodating an unlimited swarm size when unimpeded by real-time constraints. To achieve high scalability during HIL simulation, many potential optimization steps are available. For example, consider an implementation outside the simulation of a high-speed, specialized, intermediate translator that takes in raw simulation data and outputs MAVLink messages required by the Pixhawk (i.e. HIL_SENSOR and HIL_GPS messages). Such a translator would only require a stream of datagrams containing the raw FDM data from the simulation, which would take almost no time to generate, thus alleviating the need to perform such costly data translations within the simulation.

Additionally, with some careful thread optimization and task management, such translation tasks could be distributed more efficiently across multiple CPU cores and executed in parallel for increased performance. Although the default thread management within OpenEaagles is simple, easy to use, and effective in most situations, a more tailored thread management scheme could provide performance enhancements in this specific context.

One main benefit of establishing HIL simulation in OpenEaagles is preexisting interfaces to distributed simulation architectures (i.e. DIS, HLA, and TENA). Such tools not only allow sharing of simulation views across a network, but also allow multiple simulation instances to bridge together which has potential to scale the swarm simulation framework into a distributed system. Imagine multiple instances of the simulation implemented in this thesis networked together to form a larger cluster or distributed system of HIL simulation nodes. Five nodes would result in a swarm size of at least 20 UAVs, all driven by Pixhawk autopilots.

Another follow-on this research effort could take is implementing hardware interfaces to the OCA such that each hardware OCA can interact outside of the simulation with the OCAs from other UAVs (e.g. via a wireless connection, thus emulating realistic UAV peer-to-peer communications). Migrating such interactions outside the simulation continues to improve confidence that the observed behaviors will transition to real flight testing while alleviating computational requirements of the simulation environment, thus allowing those resources to be dedicated toward increased scalability.

The highly modular architecture of the proposed framework also allows for the implementation of swarm control stations (SCS) such that developers can inject parameters and swarm control algorithms mid-mission directly into the OCAs in a simulated swarm and thus transcend from simply controlling single UAVs individu-

ally (i.e. using a traditional ground control station) to flying entire swarms as single organic entities. With an infinitely configurable simulation comprised of players, systems, and subsystems, as well as customizable I/O interfaces and graphics libraries, the possibilities are endless.

5.1 Final Remarks

As technology continues to advance, UAV swarms become more feasible and relevant to real-world applications. Over the past 15 years, great strides have taken place to make UAV swarms a reality. Many have pursued the understanding and development of swarm applications, theory, and control strategies. However, most swarm behaviors remain unimplemented due to high implementation costs and regulated airspace constraints during development. This thesis shows that a swarm simulation framework can enable development and testing of swarm behaviors outside the bounds of such limitations. Bottom line: swarm development is difficult and expensive, and without a successful simulation framework to affordably and efficiently validate swarm behaviors, applications of UAV swarms will continue to exist only in the realm of theory.

Bibliography

1. F. A. Administration. Unmanned aircraft systems. Retrieved Online at URL: <https://www.faa.gov/uas/>, August 2015.
2. T. Archer and R. Leinecker. Cserial - a c++ class for serial communications. Retrieved Online at URL <http://www.codeguru.com/cpp/i-n/network/serialcommunications/article.php/c2503/CSerial--A-C-Class-for-Serial-Communications.htm>, August 1999.
3. P. Barooaha, G. E. Collinsb, and J. P. Hespanha. Geotrack: Bio-inspired global video tracking by networks of unmanned aircraft systems. In N. F. F. Jr. and V. S. Swaminathan, editors, *Bio-Inspired/Biomimetic Sensor Technologies and Applications*, volume 7321, 2009.
4. R. W. Beard and T. W. McLain. *Small Unmanned Aircraft: Theory and Practice*. Princeton University Press, 41 William Street, Princeton, New Jersey 08540, 2012.
5. J. S. Berndt et al. *JSBSim: An Open Source, Platform-Independent, Flight Dynamics Model in C++*, 2008.
6. H. Brundage. Neat algorithms - flocking. Retrieved Online at URL <http://harry.me/blog/2011/02/17/neat-algorithms-flocking/>, July 2015.
7. V. Chandrasekaran and E. Choi. Fault tolerance system for uav using hardware in the loop simulation. In *New Trends in Information Science and Service Science (NISS), 2010 4th International Conference on*, pages 293–300, May 2010.
8. P. R. Chandler, M. Pachter, D. Swaroop, J. Fowler, J. Howlett, S. Rasmussen, C. Schumacher, and K. Nygard. Complexity in uav cooperative control. In *American Control Conference, 2002. Proceedings of the 2002*, volume 3, pages 1831–1836 vol.3, 2002.
9. H. Cheng, J. Page, and J. Olsen. Dynamic mission control for uav swarm via task stimulus approach. *American Journal of Intelligent Systems*, 2(7):177–183, 2012.
10. T. O. Computing. Uav architecture. Retrieved Online at URL <http://www.twinoakscomputing.com/coredx/uv>, October 2015.
11. J. J. Corner. Swarming reconnaissance using unmanned aerial vehicles in a parallel discrete event simulation. Master’s thesis, Air Force Institute of Technology, Wright Patterson AFB, OH, March 2004.

12. R. W. Deming and L. I. Perlovsky. Concurrent multi-target localization, data association, and navigation for a swarm of flying sensors. *Information Fusion*, 8(3):316 – 330, 2007. Special Issue on Concurrent Learning and Fusion.
13. L. Evers, T. Dollevoet, A. Barros, and H. Monsuur. Robust uav mission planning. *Annals of Operations Research*, 222(1):293 – 315, 2014.
14. A. Felton-Taylor. Trial tests cost effectiveness of drone use in agriculture. Retrieved Online at URL: <http://www.abc.net.au/news/2015-06-10/drone-technology-on-australian-farms-cost-effectiveness-trial/6535546>, June 2015.
15. S. Frink. Uav swarm technology emerges to perform varied applications, August 2012.
16. C. Fuchs, C. Borst, G. C. H. E. de Croon, M. M. R. van Paassen, and M. Mulder. An ecological approach to the supervisory control of uav swarms. *International Journal of Micro Air Vehicles*, 6(4):211–229, December 2014.
17. U. Gaerther. Uav swarm tactics: an agent-based simulation and markov process analysis. Master’s thesis, Naval Postgraduate School, Monterey, CA, June 2013.
18. D. Gagne. Mavlink/qgroundcontrol. Retrieved Online at URL <https://github.com/mavlink/qgroundcontrol/tree/master/flightgear/Aircraft>, September 2015.
19. R. D. Garcia, L. Barnes, and M. Fields. Unmanned aircraft systems as wingmen. *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 9(1):5–15, January 2012.
20. P. Gaudiano, E. Bonabeau, and B. Shargel. Evolving behaviors for a swarm of unmanned air vehicles. In *Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE*, pages 317–324, June 2005.
21. P. Gaudiano, B. Shargel, E. Bonabeau, and B. Clough. Control of uav swarms: What the bugs can teach us. In *2nd AIAA "Unmanned Unlimited" Conf. and Workshop & Exhibit*, 2003.
22. H.-P. Halvorsen. Introduction to hardware-in-the-loop simulation. Technical report, Telemark University College, Postboks 203, Kjlnes ring 56, N-3901 Porsgrunn, Norway., 2011.
23. D. Hambling. The future of flight: Swarms will dominate the sky, July 2013.
24. S. Hauert, S. Leven, M. Varga, F. Ruini, A. Cangelosi, J.-C. Zufferey, and D. Floreano. Reynolds flocking in reality with fixed-wing robots: Communication range vs. maximum turning rate. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 5015–5020, Sept 2011.

25. H. Hexmoor, B. McLaughlan, and M. Baker. Swarm control in unmanned aerial vehicles. In *IC-AI*, pages 911–917, 2005.
26. D. Hodson, C. Buell, R. Suhr, D. Gehl, and L. Sines. Openeaagles simulation framework. Retrieved Online at URL <http://www.openeaagles.org/>, April 2015.
27. I. Kadar, editor. *Collaborative distributed sensor management and information exchange flow control for multitarget tracking using Markov decision processes*, volume 6968, 2008.
28. J. N. Kaiser. Effects of dynamically weighting autonomous rules in an unmanned aircraft system (uas) flocking model. Master’s thesis, Air Force Institute of Technology, Wright Patterson AFB, OH, September 2014.
29. J. L. Lambach. Integrating uas flocking operations with formation drag reduction. Master’s thesis, Air Force Institute of Technology, Wright Patterson AFB, OH, March 2014.
30. G. Lamont, J. Slear, and K. Melendez. Uav swarm mission planning and routing using multi-objective evolutionary algorithms. In *Computational Intelligence in Multicriteria Decision Making, IEEE Symposium on*, pages 10–20, April 2007.
31. L. Meier, D. Gagne, et al. Mavlink micro air vehicle communication protocol. Retrieved Online at URL <http://qgroundcontrol.org/mavlink/start>, September 2015.
32. D. Nowak, I. Price, and G. Lamont. Self organized uav swarm planning optimization for search and destroy using swarmfare simulation. In *Simulation Conference, 2007 Winter*, pages 1315–1323, Dec 2007.
33. S. O’Hara, M. Simon, and Q. Zhu. Towards distributed atr using subjective logic combination rules with a swarm of uavs. *Proc. SPIE*, 6561:65611G1–65611G10, 2007.
34. R. Olfati-Saber. Flocking for multi-agent dynamic systems: algorithms and theory. *Automatic Control, IEEE Transactions on*, 51(3):401–420, March 2006.
35. I. Ozkaya, R. Kazman, and M. Klein. Quality-attribute-based economic valuation of architectural patterns. Technical report, Software Engineering Institute, Carnegie Mellon, 2007.
36. S. Pinker et al. *The American Heritage Dictionary of the English Language*. Houghton Mifflin Harcourt Publishing Company, 5th edition edition, 2013.
37. S. Rasmussen and P. Chandler. Multiuav: a multiple uav simulation for investigation of cooperative control. In *Simulation Conference, 2002. Proceedings of the Winter*, volume 1, pages 869–877 vol.1, Dec 2002.

38. C. Reynolds. Boids. Retrieved Online at URL <http://www.red3d.com/cwr/boids/>, March 2015.
39. T. C. Rivers. Design and integration of a flight management system for the unmanned air vehicle frog. Master's thesis, Naval Postgraduate School, Monterey, CA, December 1998.
40. R. S. Roberts, C. A. Kent, C. T. Cunningham, and E. D. Jones. Uav cooperation architectures for persistent sensing. *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Defense and Law Enforcement II*, 5071:306–314, 2003.
41. J. C. Rubio, J. Vagners, and R. Rysdyk. Adaptive path planning for autonomous uav oceanic search missions. In *AIAA 1st Intelligent Systems Technical Conference*, 2004.
42. J. A. Sauter, R. S. Mathews, A. Yinger, J. S. Robinson, J. Moody, and S. Riddle. Distributed pheromone-based swarming control of unmanned air and ground vehicles for rsta. In G. R. Gerhart, D. W. Gage, and C. M. Shoemaker, editors, *Proc. of SPIE*, volume 6962, pages 69620C–69620C–12, 2008.
43. P. Scharre. Robotics on the battlefield part ii: The coming swarm. Technical report, Center for a New American Security, October 2014.
44. A. Sinha, A. Tsourdos, and B. White. Multi uav coordination for tracking the dispersion of a contaminant cloud in an urban region. *European Journal of Control*, 3(4):441–448, 2009.
45. M. P. . P. Team. *Microsoft® Application Architecture Guide (Patterns & Practices)*. Microsoft Press, second edition edition, November 2009.
46. R. Vaculin and R. Neruda. *Autonomous behavior of computational agents*. Springer, 2005.
47. T. Wang. Hardware architecture. Retrieved Online at URL http://ai.stanford.edu/~twangcat/figures/hardware_architecture.png, October 2015.
48. G. Warwick. Onr: Swarming uavs could overwhelm defenses cost-effectively. *Aviation Week & Space Technology*, April 2015.
49. S. Yili, Z. Ziyang, O. Chaojie, and P. Huangzhong. 3d scene simulation of uavs formation flight based on flightgear simulator. In *Guidance, Navigation and Control Conference (CGNCC), 2014 IEEE Chinese*, pages 1978–1982, Aug 2014.

REPORT DOCUMENTATION PAGE					<i>Form Approved</i> OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 24-03-2016		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From — To) Sep 2014 — Mar 2016		
4. TITLE AND SUBTITLE An OpenEagles Framework Extension for Hardware-in-the-Loop Swarm Simulation				5a. CONTRACT NUMBER		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Worth, Derek, B., Captain, USAF				5d. PROJECT NUMBER 16-342A		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765					8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-MS-16-M-052	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Lincoln Laboratory Massachusetts Institute of Technology 244 Wood Street Lexington, MA 02420-9108 Dr. Marc Viera, mviera@ll.mit.edu 781-981-1077					10. SPONSOR/MONITOR'S ACRONYM(S) MIT-LL	
					11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.						
13. SUPPLEMENTARY NOTES This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.						
14. ABSTRACT Unmanned Aerial Vehicle (UAV) swarm applications, algorithms, and control strategies have experienced steady growth and development over the past 15 years. Yet, to this day, most swarm development efforts have gone untested and thus unimplemented. Cost of aircraft systems, government imposed airspace restrictions, and the lack of adequate modeling and simulation tools are some of the major inhibitors to successful swarm implementation. This thesis examines how the OpenEagles simulation framework can be extended to bridge this gap. This research aims to utilize Hardware-in-the-Loop (HIL) simulation to provide developers a functional capability to develop and test the behaviors of scalable and modular swarms of autonomous UAVs in simulation with high confidence that these behaviors will propagate to real/live flight tests. Demonstrations show the framework enhances and simplifies swarm development through encapsulation, possesses high modularity, provides realistic aircraft modeling, and is capable of simultaneously accommodating four hardware-piloted swarming UAVs during HIL simulation or 64 swarming UAVs during pure simulation.						
15. SUBJECT TERMS UAV swarms, hardware-in-the-loop, simulation, OpenEagles						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U	 UU	 91	Maj B. G. Woolley, AFIT/ENG	
					19b. TELEPHONE NUMBER (include area code) (937) 255-3636, x4618; brian.woolley@afit.edu	