April 2019

Diagnosing Robotic Swarms (Dr. Swarm)

Alexandra Wheeler
Worcester Polytechnic Institute

Erika Schroder Snow
Worcester Polytechnic Institute

Jerish Benjamin Brown
Worcester Polytechnic Institute

Josiah Daniel Boucher
Worcester Polytechnic Institute

Diagnosing Robotic Swarms (Dr. Swarm)

A Major Qualifying Project Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for
the Degree of Bachelor of Science

By
Josiah Boucher (RBE)
Jerish Brown (RBE/CS)
Erika Snow (RBE/CS)
Alexandra Wheeler (RBE/CS)

Submitted on April 25th, 2019
To Professors:

Carlo Pinciroli
Lane Harrison

Abstract

Troubleshooting a robotic swarm can be a daunting task due to large quantities of information to sift through and many potential sources of problems. Currently there are no widely adopted swarm diagnostic systems. We developed Dr. Swarm, a mobile application which combines state-of-the-art AR technology and existing visualization techniques to create a new kind of diagnostic tool for swarm robotics. Dr. Swarm enables developers to expose the behavior of swarm systems through intuitive visualizations and assists with troubleshooting swarm applications.

Table of Contents

Table of Figures

Introduction

Swarm robotics has broad applications and immense potential. A robotic swarm is a large number of robots interacting to create a collectively intelligent system [1]. Unfortunately, diagnosing issues that occur with robot swarms during development is difficult. This is because there are so many potential sources of error; all of the problems that occur with a single robot exist for robot swarms, such as mechanical failure, sensor malfunction, or faulty code. Additionally, robot swarms communicate with one another, causing every additional robot to exponentially increase the possible sources of error.

Currently, there are no tools available to help diagnose the source of an error within a swarm beyond conventional integrated development environments (IDEs). This leads to swarm developers spending a large portion of time digging through low-level data to find what went wrong. The goal of Dr. Swarm is to enable swarm developers to more easily diagnose the behaviors of swarm systems. Dr. Swarm is a general-purpose diagnostic tool for swarm robotics that provides augmented reality visualizations to expose relevant data from the swarm. Seeing this information in real-time visualized above the robot allows developers to quickly identify issues as they happen and provides a starting point to fix the problem.

Background

Swarm Robotics

A robotic swarm is a large number of robots interacting to create a collectively intelligent system [1]. These systems operate autonomously in large numbers and without centralized control. In an ideal swarm system, the individual robots are dispensable and interchangeable [2]. Examples of robots designed to be part of a swarm are the Kilobot and Khepera IV. Both of these robots house several sensors and can communicate with other robots. Swarms typically utilize decentralized control in order to define the tasks which the robots work together to complete. This allows swarms to work at smaller problems and have more reliable decision-making. Common swarm tasks include flocking, foraging, and collaborative manipulation. These tasks can vary in difficulty, output, and robot requirements (i.e., quantity and sensors).

Many industries are putting swarm robotics to use, from health care to automation. Within these industries, there are four main areas where swarm robotics is useful. The first is tasks that cover a large amount of physical space. Since swarm systems are able to scale up in size and abstract the data collection from the individual robots, they are able to create a distributed system that can complete tasks such as surveillance over very large areas. Secondly, swarms can also complete tasks that are too dangerous for humans or more expensive robots. Robot swarms are envisioned to be formed by dispensable, inexpensive, and easily replaceable units, making them ideal for situations involving risk. The third area is “tasks that scale up or scale down in time” [1]. Robotic swarms can resolve these issues more easily than other systems due to their scalability; as a problem, such as an oil spill, gets worse over time, the system can easily add additional robots to fit the new scope. Finally, swarm systems excel at tasks that are repetitive and redundant [1]. Current research applies these strengths to aerial swarms, patrolling swarms, foraging swarms, and much more.

Open Problems in Swarm Robotics

Despite the potential shown in this relatively new field, there are still many problems that hamper the advancement of swarm robotics. One problem the field currently faces is how expensive a large swarm can be. Even relatively low-tech swarms comprised of potentially hundreds of robots require a significant investment to perform research. Error propagation is another significant obstacle that swarm researchers face. Even with the simplest of algorithms, errors have a tendency to cascade through both space and time. This can be difficult to correct when dealing with large swarms. As the size of a swarm increases, its behavior becomes more and more unstable as errors amplify through the swarm [3]. Error propagation is an open problem in swarm robotics and the research community is currently devoting effort to devising more robust solutions.

Another problem with swarm robotics is the amount of information gathered. Although the individual robots that make up a swarm tend to be simplistic with only a few sensors, an entire swarm's worth of sensors provides a significant amount of data. This sort of data is most useful when evaluated in real-time, since "[a robot's] internal state is normally meaningful only if considered alongside with the physical environment" [4]. Compounding on that is information about how robots interact with each other, which is important for understanding the behavior of the entire swarm. Generally, this information is unintuitive and must undergo processing in order to summarize the raw sensor data and make it more comprehensible.

Debugging a robotic swarm is more challenging than debugging a single robot, and very few tools exist to mitigate this. Problems with individual robots typically lie in either the hardware or the software. However, things get more complex with a robotic swarm, as "the behavior of an individual robot in a swarm robotic system is a product not only [of] its hardware, software, and interactions with its environment, but also its interactions with other robots" [5]. Due to large quantities of information to sift through and many potential sources of problems, troubleshooting a robotic swarm can be a daunting task.

Augmented & Mixed Reality

Augmented reality (AR) is a versatile technology growing in popularity. This field involves displaying digital graphics in tandem with the real world, typically by overlaying useful information or utilizing a user interface. Uses of this technology have expanded since its origin in the mid 90's, and now has applications in areas including education, tourism/exploration, and entertainment/gaming [6]. AR provides a convenient and seamless method to display important information for users of many systems, including swarm research applications.

ARDebug is an open-source AR tool used for debugging robotic swarms [5]. This tool displays an augmented video feed of robotic swarm systems which provides customizable overlays. It also equips users with the ability to map data to tables and graphs, providing live information about various system components. Despite limitations to a 2D plane of wirelessly connected robots, ARDebug is an exemplary model of a tool used for effective swarm debugging using augmented reality [5]. Other systems, however, have shown promise with robotic systems operating in three dimensions.

Mixed reality (MR) allows AR to interact with physical robots, producing many benefits including simplified debugging and scaling up swarms [7]. Hoenig et al.'s [7] research applies this use of AR with some success. MR serves as a stepping stone from simulation to physical operation. Using the game engine Unity, Hoenig et al. [7] simulated human movement patterns in tandem with a physical quadcopter. This enabled safe and easy testing of an otherwise dangerous and unpredictable task.

Using AR and MR has the potential to solve many common problems when developing robotic swarms. For example, simulating robots in the swarm negates the expense of purchasing large quantities of robots, and allows developers to test in a controlled environment before moving on to potentially problematic robots. Additionally, this simulation enables developers to add virtual sensors, further reducing costs and time [7].

Current Debugging & Profiling tools

Debugging tools help developers test programs and identify bugs. Tools like GDB allow developers to run their program and stop at specified points or conditions. Developers then see what is happening in the program by examining data, the call stack, memory, and more. Debugging tools also allow developers to experiment with the behavior of their programs. Other tools exist to examine the performance of programs; these profiling tools allow developers to collect data such as runtime, memory, and other performance statistics. Integrated Development Environments (IDEs) combine debugging, profiling, and build tools with a text editor into a single program. Having all the necessary development tools in one spot is convenient for developers, enabling the various tools to work in tandem to make debugging easier.

In the context of robotics, debuggers and profilers both play important roles. Robotic systems typically use custom software to assist in debugging issues related to sensors and actuators. For example, the Robot Operating System (ROS) [8] has *rviz*, which allows developers to visualize the robot and relevant sensor data. Many other tools exist in the ROS ecosystem that help debug and profile robot performance, such as *rqt_bag*, *rqt_plot,* and *rqt_graph*. These are profiling tools, as they exist to measure the performance of the program by examining robot behavior and sensor data. Developers sometimes also use traditional development tools, like the PyCharm IDE, to create ROS programs.

No comprehensive set of tools exists for developing and debugging swarm programs, as it is a relatively nascent field. Difficulty arises when debugging swarm robotic systems due to the large number of agents involved in the system [5]. Before deploying code to real robots, simulating the behavior of the swarm in software can help developers catch problems early on. Swarm developers use existing tools such as ROS to simulate swarm systems, but developers often work with robots that do not support ROS. In these scenarios, developers create custom solutions to simulate and debug their systems, such as simulations in the Unity game engine [9] or tools like ARDebug.

Research institutions across the world use ARGoS, a general-purpose multi-robot simulator. Designed from the ground up with accuracy, extensibility, and performance in mind, it allows developers to create plugins that support different robot platforms, as well as the simulation of large numbers of robots [10]. Additionally, ARGoS supports Buzz, a programming language designed specifically for swarm robotics. Buzz is extendable, allowing the language to work on any robot platform. However, Buzz does not have a comprehensive debugging/profiling tool suite available, making it more difficult to use than similar tools. Using ARGoS and Buzz together enables swarm developers to inspect data within the robot programs at runtime. This combination provides developers with a robust suite of swarm-specific development capabilities.

Design

The goal of this project is to enable swarm developers to more easily diagnose the behaviors of swarm systems. We designed an augmented reality (AR) diagnostic tool for swarm robotics called Dr. Swarm. This tool allows swarm developers to see relevant information about their robots visualized in the real world. This section discusses the initial design goals and structure of Dr. Swarm.

In order to achieve this goal, we identified the following objectives:

- Gather data from many robots.
- Provide intuitive visualizations conveying information about swarm robots to developers.
- Easily integrate with existing swarm projects.
- Allow compatibility with different devices simultaneously.

Before visualizing information about robots within a swarm, Dr. Swarm must gather data from the robots. We chose to focus on data from variables stored in robot's memory. Dr. Swarm uses this data to display visualizations, allowing developers to intuitively understand what is going on in their system. We designed Dr. Swarm with the intention that it could be used regardless of the system with which it is being integrated. Dr. Swarm should work on different types of devices such as AR headsets, mobile phones, and tablet devices. Additionally, we wanted the application to be capable of running simultaneously on multiple devices, allowing groups of developers to have multiple perspectives.

Example: Foraging Experiment

In order to understand the goals of this project, consider an experiment where a swarm is implementing a foraging algorithm. In the experiment, the robots start at their home base, explore their environment in search of food, and bring any they find back to base. In this example, a user selects the bar graph visualization from the user interface. Each bar graph hovers above a robot and displays the amount of food it has gathered, updating its values in real time. With this information present as the experiment runs, the user is able to determine that a specific robot is underperforming compared to the others. The user decides to add some new visualizations that specifically target the underperforming robot and is then able to determine that the robot’s battery is low. This is just one example of how Dr. Swarm can reduce the time it takes to troubleshoot a problem.

Overall System Structure

The overall structure of Dr. Swarm is modular. Isolating the parts of the system allows different application developers to easily replace components to better fit their needs. Between any two components, communication is only one-way; this furthers the modularity of Dr. Swarm, ensuring that replacing one component requires change from only one point of communication.

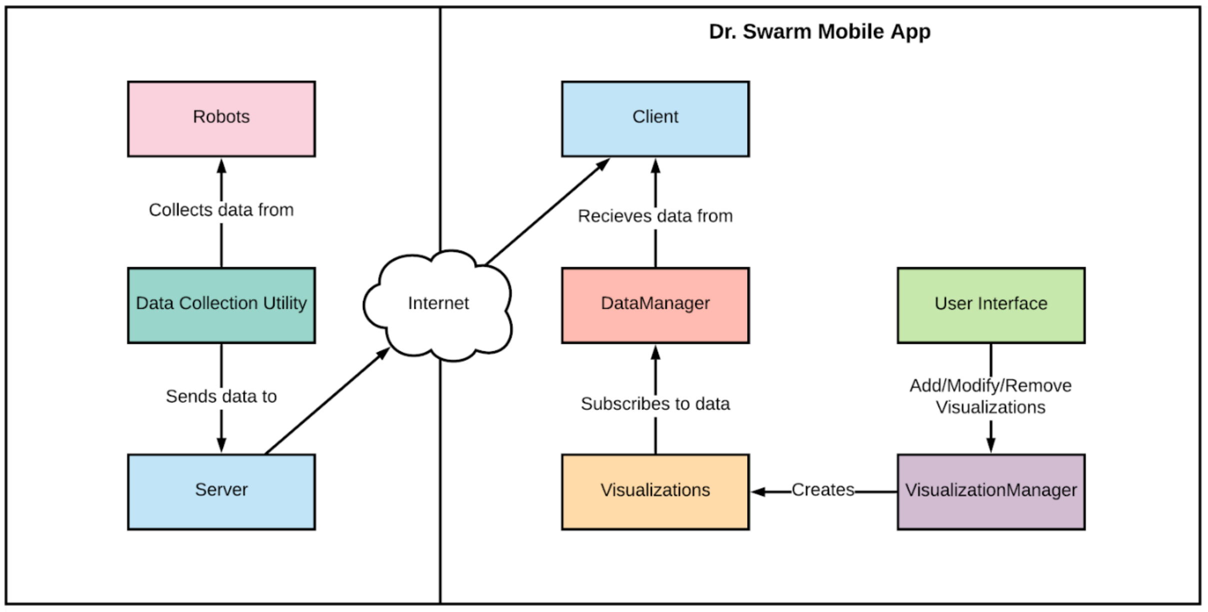

Figure 1: Dr. Swarm System Structure

The system starts with the collection of data from the robots. Data collection can happen in one of two ways: centralized or decentralized. In a decentralized manner, each robot reports its own data. In a centralized manner, there is a single entity (typically external to the swarm) which is in charge of collecting data from the robots. Although swarms typically use decentralized methods over centralized ones, we chose to support a centralized approach to data collection because it was simpler to implement. This takes the form of a single server, which sends data previously gathered from the robots to connected instances of the Dr. Swarm mobile application.

In order for the mobile application to keep track of this data, the system needs to store the data and make it accessible to different parts of the application. Visualizations specify the robots that act as data sources for the visualization and display the requested data streams from each of these robots. Users must be able to create, edit, and delete visualizations on-the-fly, which makes visualizations another kind of information to keep track of. This outlines the two main systems that drive Dr. Swarm, the `Data Manager` and the `Visualization Manager`.

The `Data Manager` is responsible for storing data received from the server and making it accessible to the rest of the application. The `Data Manager` provides information about what data has been received from the robots, as well as exposing the data itself. Visualizations take data from one or more robots cataloged in the `Data Manager` and display it in AR in the form of a line graph, bar graph, pie chart, range indicator, or a map indicator. The `Data Manager` also provides the ability to know when data changes, so visualizations can update in real time.

The `Visualization Manager` is in charge of maintaining data about the visualizations and adding and removing visualizations. Since there are multiple ways to create visualizations, any part of the application must have access to the `visualization Manager` at any time. When the `Visualization Manager` receives a new visualization to add, it is in charge of attaching the visualization to each corresponding robot and creating the graphical elements of the visualization. Likewise, when a user deletes a visualization, the manager is responsible for removing the visualization from the display and removing the data associated with that visualization.

We designed Dr. Swarm to support two methods of visualizing data in AR: graph-based visualizations, and indicators. Graph-based visualizations consist of robot data displayed on a graph that exists in AR. We designed the application to support three types of 2D graphs: pie charts, bar graphs and line graphs. Indicators are 2D objects, such as an arrow representing a vector or a symbol representing the internal state of a robot.

User Interaction & Control

We designed a user interface for Dr. Swarm in order to give users easy access to the application's main features. The main menu serves as a central hub for control of the application. The menu contains tabs for controlling different parts of the application, including visualizations, the communications display, robot tagging, and other miscellaneous options. The most important tab is the visualizations tab, which allows users to add and remove visualizations. This tab displays a list of added visualizations with an image preview for each one. There is also a checkbox for each visualization; toggling that checkbox will display or hide that visualization above the robots.

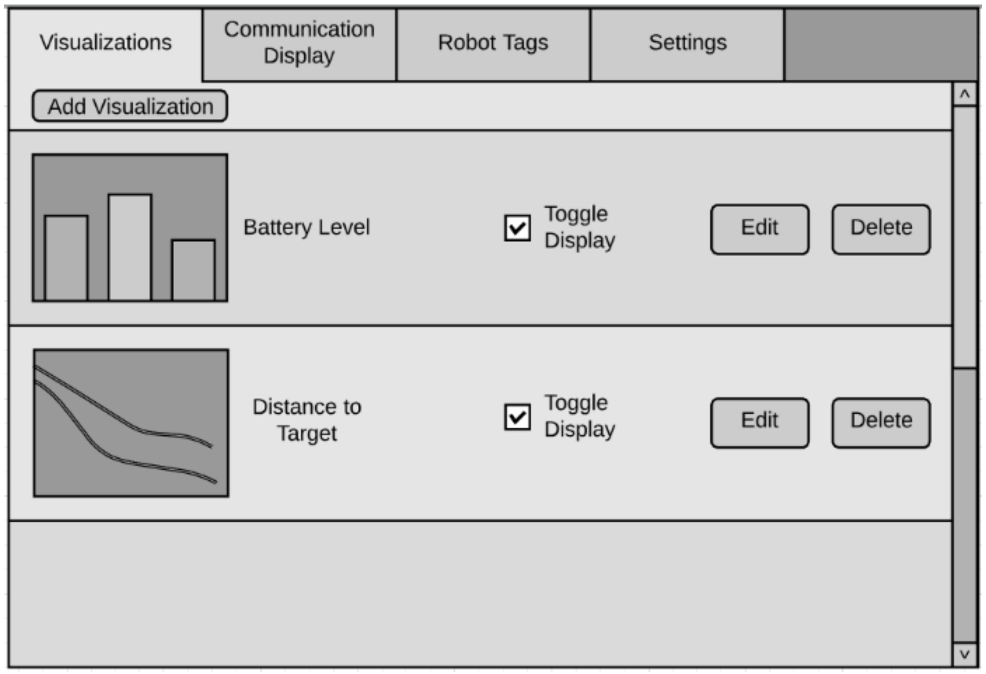

Figure 2: Initial Design of the Visualization Menu

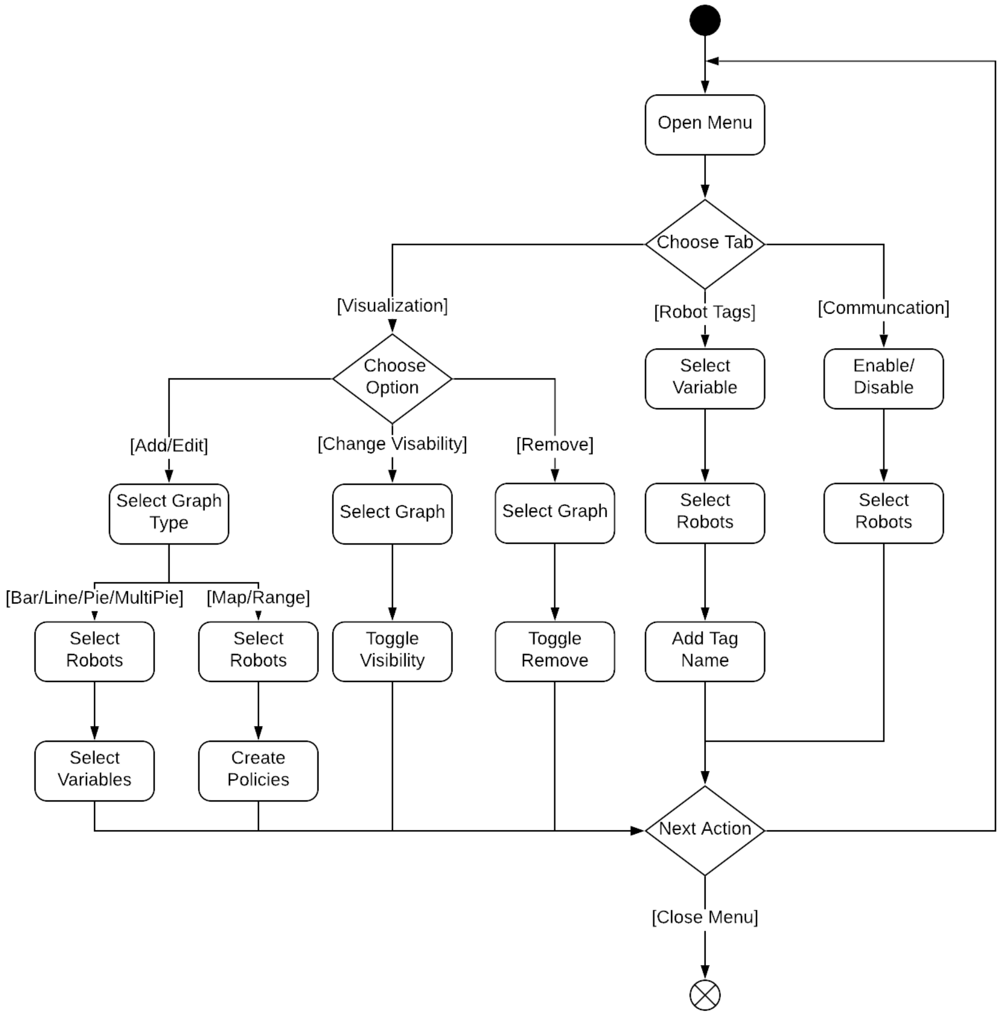

Figure 3: General User Activity Diagram for Visualization Creation & Modification

The "Add Visualization" button triggers a wizard UI which walks users through the process of adding a new visualization. This process is shown in Figure 3. First, the user selects a visualization type. This can be one of the three graph types (bar, pie, line), or a map or range indicator. Second, the user selects which robots to source the data. Finally, the UI transitions to a screen that displays a filtered list of data for the user to select from; this list excludes data that is not on the intersection of the previously selected robots. Different kinds of visualizations support different numbers of data stream and robots. For example, a line graph, supports two data streams (one per axis) for many robots.

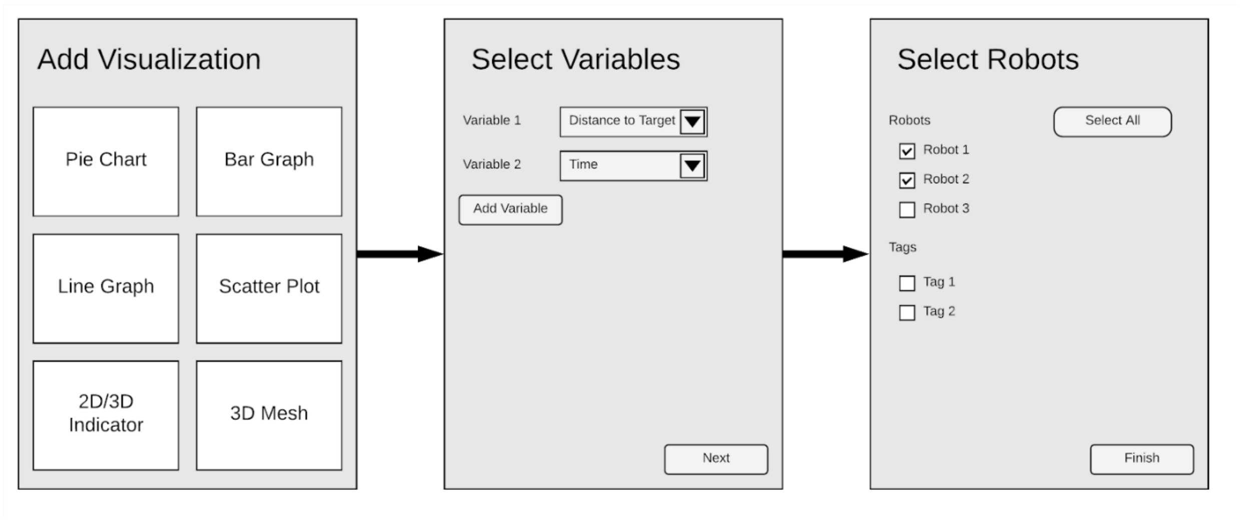

Figure 4: Add Visualization Workflow

Example: Foraging Experiment

From the earlier example of the foraging experiment, a user could use this menu system to add a line graph. The user selects the line graph type and the robots that the graph would be over. The user then selects the data stream that is distance to a food source. Then, over each selected robot, a line plot displays the distance from a food source over time.

Implementation

The first step to creating Dr. Swarm was finding a framework for developing augmented reality (AR) applications. The WPI NEST Lab had a previously existing application for human-swarm interaction using Vuforia. Vuforia is an AR framework that runs inside the Unity game engine. We determined that using Vuforia and Unity would allow us to rapidly prototype ideas which we could expand into our finished product. Furthermore, Vuforia and Unity support exporting to many platforms, such as AR headsets, Android devices, and iOS devices. This cross-platform support was important because we prioritized accessibility for swarm developers regardless of their platform. One major limitation is that Vuforia currently supports tracking of up to only five image targets. This means that Dr. Swarm can display only five visualizations at the same time on any one device.

We developed our application to accommodate the setup of the NEST lab at WPI. The lab allowed us access to a small swarm of Khepera IV robots, and a larger swarm of Kilobots. The Kilobots were too small to attach tracking images that would work with Vuforia. We decided to focus on the Khepera IV robots. They had already been set up for tracking with Vuforia for use in another application being developed in the NEST Lab, which made them easy to integrate with our application. The Khepera IV robots were controlled using ARGoS and a VICON motion capture system. The VICON system tracks small markers on the robots and feeds their position into ARGoS. In this setup, ARGoS is used as a centralized controller which coordinates the robots.

Data Management

The Dr. Swarm mobile application needs to keep track of the data received from every robot. There are two sets of clients, who will use the data in various ways: the visualizations and the user interface. As previously mentioned in the Design section, visualizations need to take data and display it in AR. This means that visualizations need to know not only what data exists, but also the type of the data and what the current value is. Visualizations also need to know when this data changes, so it can update accordingly. Additionally, the “Add Visualizations” menu in the user interface needs access to what data exists, so it can show the user what data they can visualize. These requirements drove the design of the `Data Manager`, specifying the following requirements:

1. There must be one location for reading and writing all data.
2. Data must be indexed by which robot it belongs to.
3. All data must be accessible at any time by all clients.
4. Clients may choose to be notified when data changes.

To satisfy these requirements, we decided that the `Data Manager` would be a singleton object that stores the association between robots and their data. Clients can query the `Data Manager` with the unique identifier of a robot, and it will return an object containing information about the robot. The robot object contains its identifier, a color, and the data associated with the robot. These features satisfy the first three requirements, but not the fourth. To allow clients to be notified when data changes, we decided to use the observer software design pattern. We chose to implement the pattern at the data-level. Clients can subscribe to individual pieces of data which notifies the subscriber when the data changes.

The decision to use the observer pattern drove the development of the Dr. Swarm application as a whole. We chose to implement some features of Reactive Programming, a paradigm where data takes the form of streams instead of immediate values. Specifically, we decided that the flow of all data in Dr. Swarm should be reactive. This allows for high levels of modularity and loose coupling between classes.

Dr. Swarm needs to keep track of many different types of data, including numerical values (`floats, ints`), textual values (`strings`), and boolean values. Each piece of data must have a name associated with it, have the name of the data stream retrieved from the robot, and be stored inside the robot object. This could not be accomplished with a simple dictionary, because a dictionary maps keys of one type to values of another type. We created a class called `VariableDict`, a data structure which maps a string key to a value of any type. Each key is associated with one value of a single type. This class uses two nested dictionaries; the outer dictionary associates data types to the inner dictionary. The inner dictionary maps the name of the data stream to the stream itself. This allows Dr. Swarm to have a one-to-one mapping of names to values, but the type of the values can differ between different pieces of data.

Visualizations

Each visualization is broken up into two classes: a class that handles the functionality of the visualization, and a container class that handles displaying the visualization in Unity. Both the functionality classes and the container classes implement their own interfaces, which allows us to use the same methods on different types of visualization objects. In addition to increasing the uniformity of our code, these interfaces improve readability and provide a straightforward way of adding other visualization types in the future. We considered changing these interfaces into abstract classes. We felt abstract classes would be a better design than the interfaces because each of the functionality classes and each of the container classes share much of their code, including both fields and methods. However, we left the interfaces in place due to time constraints during implementation.

We implemented four types of visualizations for Dr. Swarm. Below is a table summarizing these visualizations. The data streams supported, and robots supported are in reference to a single visualization appearing above one robot.

Visualizations	Subtypes	Data Streams Supported	Robots Supported
Pie Chart	Multi-Robot	1	1+
	Multivariable	1+	1
Indicator	Range	1	1
	Map	1 - 3	1
Bar Graph	-	1+	1+
Line Graph	-	2	1+

Figure 5: Visualizations by Quantity of Variables and Robots/Tags Supported

Pie Chart Visualizations

Single variable pie charts support a single data stream across multiple robots. The visualization above the robots selected for a single variable pie chart will contain data exclusively from the robot below.

Multivariable pie charts support multiple streams of data from a single robot. With this visualization, multiple robots can be selected, but when they are displayed above the robot in the application, the information will be tailored to that specific robot. This differs from single variable pie charts, where the visualizations above the robots pertain only to the robot below.

Multivariable Pie Chart Foraging Example: Time in each State

During the foraging experiment, the user wants to know how much time each robot spends in each of three states: exploring the environment, collecting food, returning to base. They create a multivariable pie chart visualization that tracks the percentage of time in each state for each robot. The user notices that some robots spend the majority of their time collecting food, and very little time exploring the environment, and hypothesizes that those robots are prioritizing food sources close to the base.

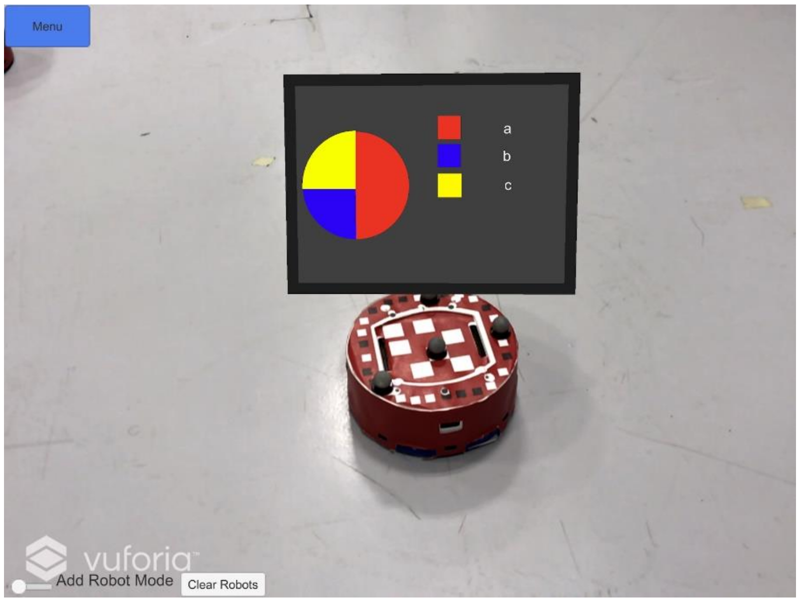

Figure 6: Multivariable Pie Chart

Pie Chart Foraging Example: Food Collected

The user wants to explore their hypothesis by tracking which robots have collected the most food. To accomplish this, they add a pie chart visualization that tracks the total amount of food collected by each robot. When the experiment is run, the user notices that the robots that have collected the most food stick close to the base and are also the same robots who spend the most time collecting food. The user decides to modify the foraging algorithm to prioritize exploring the environment close to the base in order to improve the efficiency of the swarm.

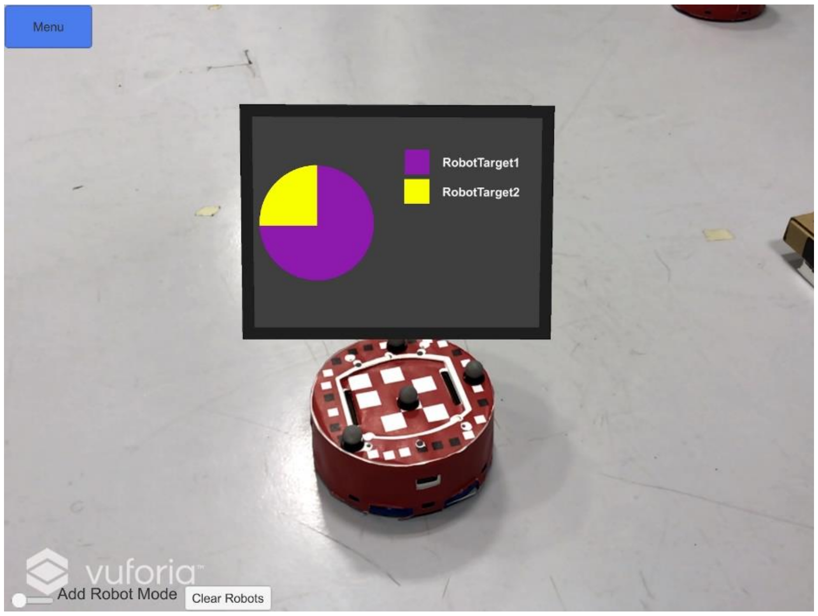

Figure 7: Pie Chart

Indicator Visualization

The indicator visualization supports any number of robots, and a single data stream for each of their features. Currently, there are two types of indicator visualizations: range indicators and map indicators, each with different kinds of features. Range indicators allow data streams to be associated with the color and the shape of the visualization. Map indicators require a shape to be specified during creation, but a data stream can be associated with its color, rotation, and fill amount. With this visualization, multiple robots can be selected, but when they are displayed above the robot in the application, the information will be tailored to that specific robot.

Indicators, of any kind, utilize policies to define the behavior of the visualization. Policies associate a data stream with a feature of the indicator. Range indicators use range policies to define a range of values for which the indicator should be a certain color or a certain shape. A range indicator can have as many of these policies as the user needs in order to cover the desired ranges and behaviors of the tracked data streams. Map indicators use map policies to define which data stream should be mapped to a feature. A map indicator can currently have a maximum of three map policies since there are only three features defined for a map indicator.

Range Indicator Foraging Example: Robot State

During the foraging experiment, the user wants to know what state a group of robots is in at any given time. Specifically, they want to see when each robot changes its state. Conveniently, the robot has a state data stream. When the data stream is equal to zero, the robot is exploring its environment. When the data stream is equal to one, the robot is gathering food from a known source. And when the data stream is equal to two, the robot is returning to its base. The user creates a range indicator and specifies three range policies. When a robot's state is zero, the indicator will display a red circle. When the state is one, the indicator will display a blue square. And when it's two, the indicator will display a green triangle. With this visualization in place, the user is able to clearly see that a specific robot never escapes the exploration state even though its sensors should have discovered multiple sources of food during the experiment.

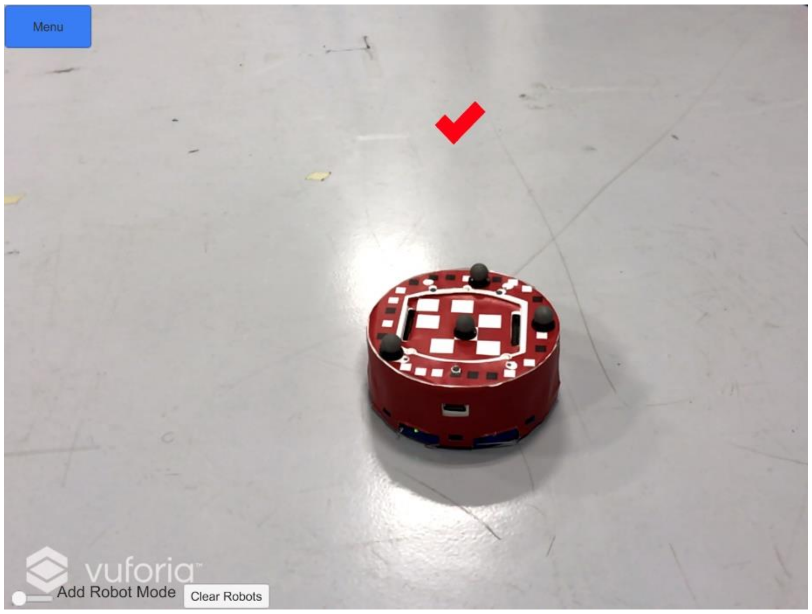

Figure 8: Range Indicator

Map Indicator Foraging Example: Battery Level

The user notices that some of the robots run out of battery much faster than others. They decide to keep an eye on the battery level of each robot through a map indicator. The user sets up the indicator so that the battery voltage is associated with the fill amount of a circle. As the voltage drops, the circle wedge gets smaller. Every robot in the swarm starts out with a full charge. However, as the experiment continues to run, it becomes apparent that some of the robots lose their charge much faster than others. By tracking these robots, the user is able to determine that they are traveling farther than most of the other robots. The user modifies the swarm behavior to equalize the distance each robot travels, giving the swarm a longer life-span.

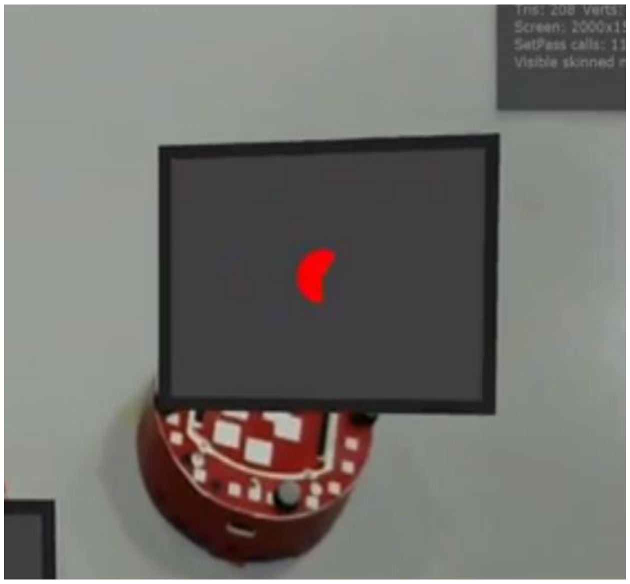

Figure 9: Map Indicator

Bar Graph Visualization

Additionally, we created a bar graph visualization, which supports any number of variables on any number of robots. When the visualization is displayed in the mobile application, a legend below the graph shows the color associated with each data stream.

Bar Graph Foraging Example: Proximity Sensor

Even though the user modifies the foraging algorithm to even out the amount each robot travels, some of the robots' battery levels are still dropping faster than others. The user notices that these robots seem to be moving past sources of food instead of entering the gathering state.

The user decides to add a bar graph visualization to these robots that shows data from their sonar sensors. In this way, the user can see how the robots are perceiving obstacles in their environment. From this visualization, the user is able to determine that some of the sonar sensors are not operating correctly.

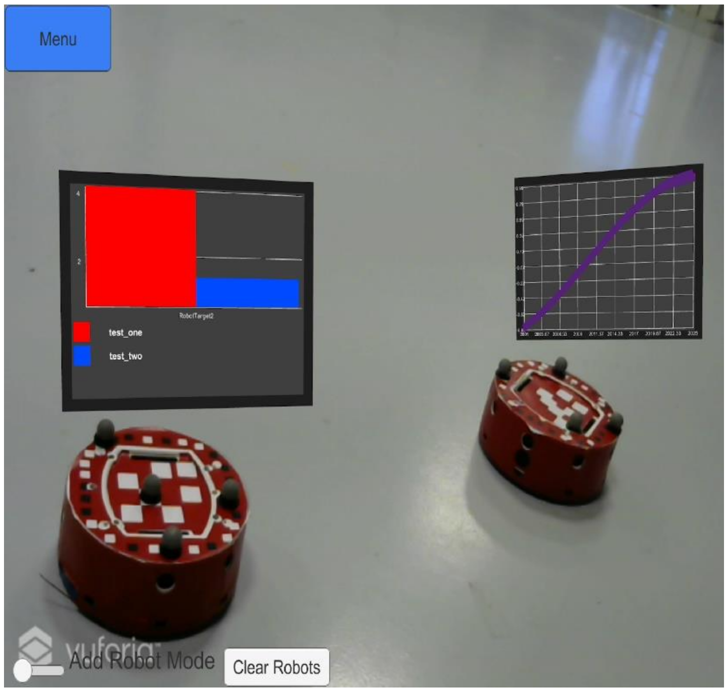

Figure 10: Bar Graph

Line Graph Visualization

The last type of visualization we developed is a line graph, which supports a variable for the x axis and another for the y axis. Any number of robots can be associated with an instance of this visualization, and the mobile application will display the line graph above each robot.

Line Graph Foraging Example: Proximity Sensor

The user now knows that some of the sonar sensors are not operating correctly, but they want to see the data from these sensors over time in order to better understand how they are operating. They create a line graph visualization on the robots that plot the data from the malfunctioning sonars. The user is able to determine that the sensors are able to accurately detect objects that are farther away, but they lose sight of the object when it is brought within a few inches of the sensor.

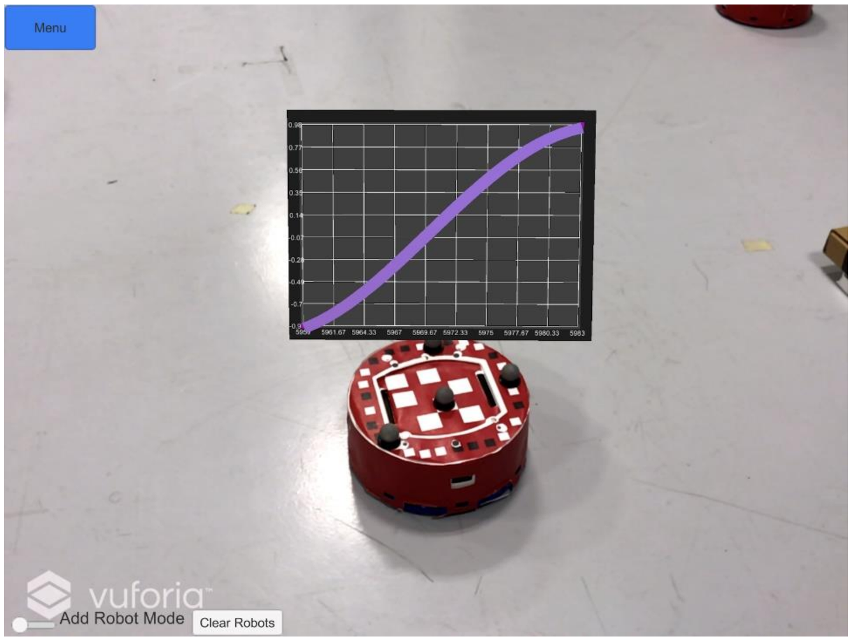

Figure 11: Line Graph

Visualization Management

One main goal for Dr. Swarm is to display many different visualizations at the same time. Visualizations need to be created in real-time by user input. Originally, we had planned to support pre-defining visualizations to be displayed and creating visualizations over the network, but these features were not implemented. Regardless, these drove the design of the `Visualization Manager`, a central system where visualizations can be added, removed, and edited. Similar to the `Data Manager`, any client within the application can have access to the `Visualization Manager` at any time. This allows for the User Interface, described in detail in the following section, to simply acquire the instance of the `Visualization Manager` and then call the `AddVisualization` method to add new visualizations. Future work could easily integrate the `Visualization Manager` to implement new features such as pre-defined visualizations or adding through the network.

While designing the visualization system, we decided on two main features the system must have. First, visualizations need to be able to be added, removed, and edited in real time. Second, robots needed to be able to easily identify visualizations associated with themselves. In order to create these features, we again turned to Reactive Programming. Representing a visualization as a stream of data allows clients to subscribe to a specific visualization to get updates about when it is added, edited, or removed. This also reduces overhead by only notifying clients when a visualization changes, instead of the clients polling for updates about visualizations. Next, we decided that the `Visualization Manager` would keep track of the robot-to-visualization associations manually. This too would be a stream, so clients could subscribe to their own stream of visualizations.

As mentioned in the visualization section, container classes are responsible for taking data and displaying it to the user. Since a robot can have multiple associated visualizations, we needed to create a class that would aggregate all the containers for the visualizations associated with each robot. We called this the `Visualization Window`. This class displays a panel behind containers to help contrast the visualizations from the real world, and handles switching between multiple visualizations. A `Visualization Window` lives on each tracked robot object and subscribes to the list of visualizations for that robot. When a visualization is added, it creates a container for that visualization type and adds it to the scene. Likewise, when visualizations are removed it deletes the container from the scene.

User Interface

The user interface (UI) is broken up into several Unity panels on a single Unity canvas. We chose this method because it makes it easier to interact with the AR elements and to organize the panels for modification. Different panels have their own functions, from displaying current visualizations to adding robots to a visualization. We originally wanted to have one class containing the majority of our helper functions, but when the class needed to be added to the panels, there were issues with the individual UI components. This led to individual panels having their own script to determine what and how the content is displayed on the panel. However, most panels also interact with the `UI Manager`.

The main component of the user interface is the `UI Manager`. This is a singleton class that manages the UI by processing the big decisions and interacting with the `Data Manager` and `Visualization Manager`. The main functions of this class are adding and editing visualizations, as these functions are spread across several UI screens.

The majority of the Unity panels are dynamic, displaying different elements depending on what has been previously done. Unity provides dynamic elements through scripting and utilizing prefabs. We chose to use Unity prefabs, as they allow us to merge several game objects into one, and then replicate that new game object many times. This process is significantly easier than adding every element individually and provides a level of consistency. For example, when adding a range policy, every dropdown and text input are in the same spots. Additionally, it makes it easier to query each prefab for what the user has selected, and to add listeners to different elements.

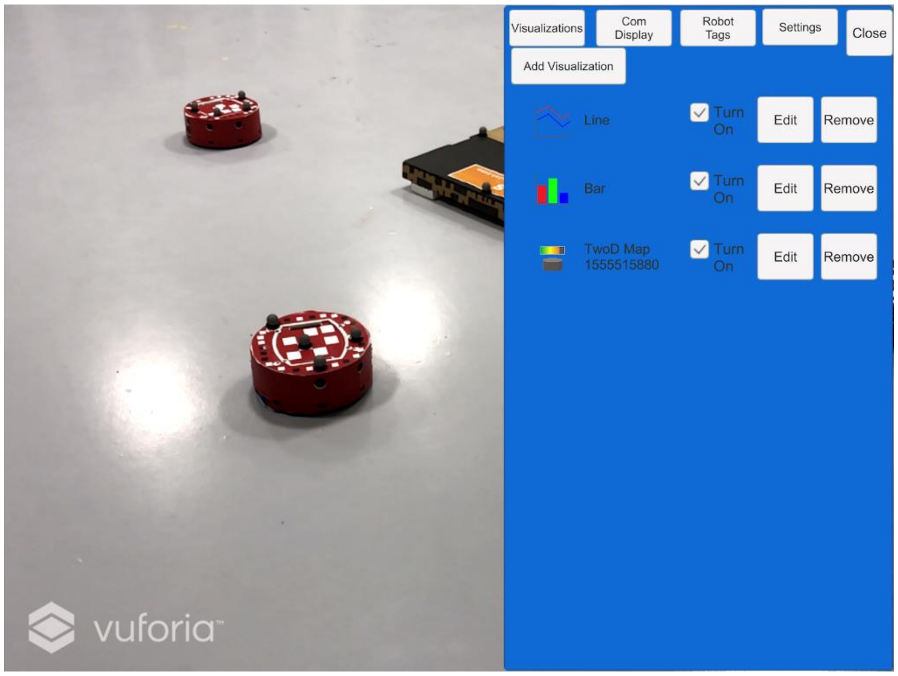

Figure 12: Main Screen Menu

We created a wizard UI for adding and editing visualizations. This provides users with a precise way of creating and editing while also reducing potential errors. The adding and editing processes are essentially identical for the user. The first step is to select the type of visualization, which determines if the user adds policies or data streams in subsequent steps. Next, the user adds desired robots to the visualization by individually selecting them or hitting “Select All”. Then, the user can add the data streams or policies. If a user selects line, pie, or bar, they can then add the number of data streams they want. These data streams are an intersection of the total data streams of the robots they previously selected. However, if a user picks map or range, they can add one or more policies. Once a visualization has been added, it can be removed via the main screen for visualizations.

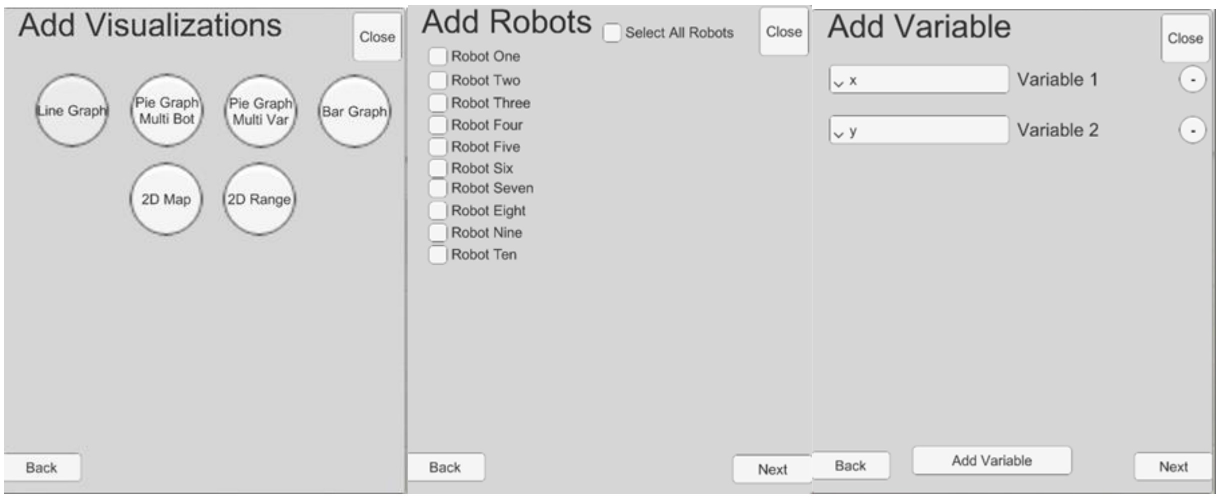

Figure 13: Process for Adding a Visualization

We also implemented a process for creating tags for robots. This process makes it easier and faster to add specific groups of robots to visualizations. A user can create a tag that contains a name and a set of robots. Then, when a user is adding robots to a visualization (whether adding or editing), the user can select a tag to add the robots under that tag. This will add the set of all robots the user had selected, including any that were previously tagged. Figure 14 shows the two screens responsible for the creation and display of tags.

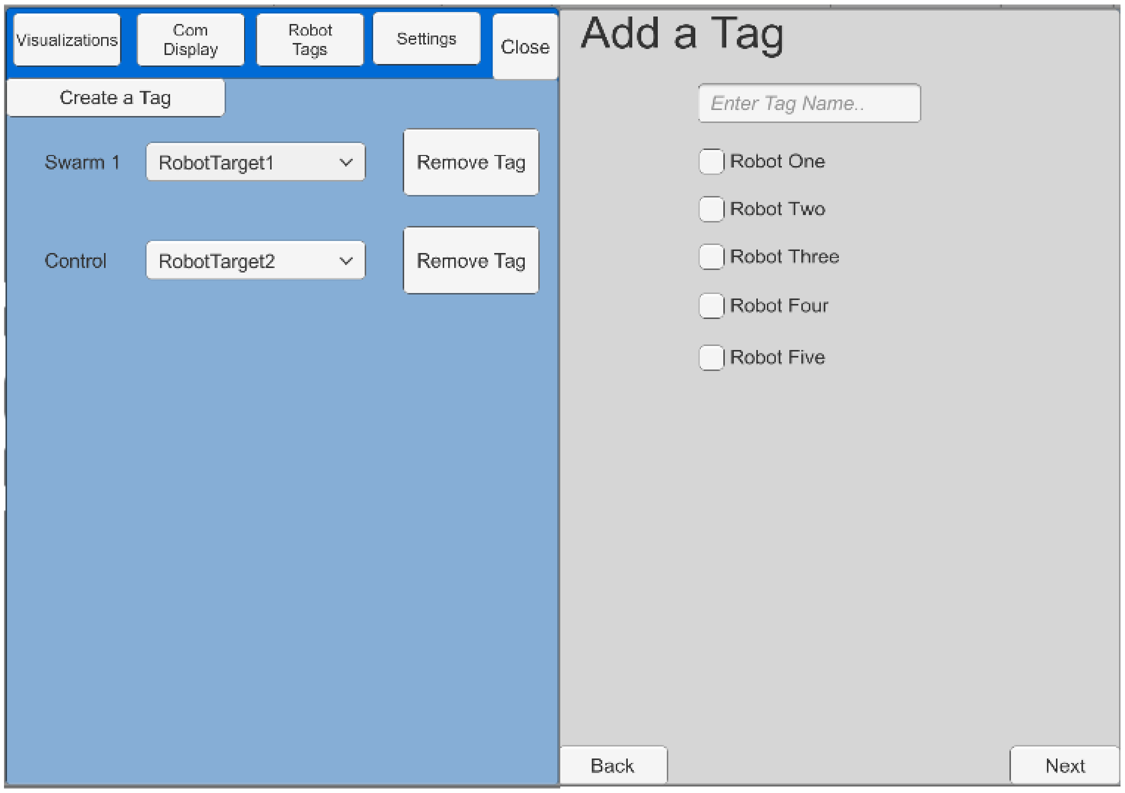

Figure 14: Tag UI

There are a few other features added to the UI to make the application more user friendly. The first miscellaneous feature is that a user is able to select robots by tapping on them on their mobile device. Once a user has clicked on a robot, those robots will appear checked when adding a visualization or creating a tag. This allows a user to add robots to a visualization without knowing the name of the robot. This feature can be toggled on and off. Additionally, we found that the map and range indicators could be difficult to see due to their size. Because of this, we added the ability to remove the background from the indicators when the visualization is displayed. At the moment this feature is only implemented when a user switches between visualizations by tapping on a visualization above a specific robot on the mobile device.

Variable Publisher

The `Variable Publisher` is a separate application which aggregates data from the robots within a swarm. As discussed in the design section, this could be accomplished in either a centralized or decentralized manner. We chose to use the centralized method, where one program collects data from all the robots, and broadcasts that information to instances of the Dr. Swarm application. We considered allowing each robot to broadcast its own data but decided that the centralized method was simpler to implement.

The Dr. Swarm application is designed to accept a JSON message from the `Variable Publisher`. JSON is a widely used human-readable data interchange format. Its ease of use due to the abundance of parsing libraries available for many languages made it a clear choice for us. Currently, the protocol only encodes the data associated with each robot. In the future, adding new features could be accomplished simply by adding new tags into each JSON packet.

```
{
    "variables":{
        "RobotTargetN":{
            "variable1":1.0,
            "variable2":2.0
        }
    }
}
```

Example of a JSON packet

Developers can create their own implementations of the `Variable Publisher` to use with their own robot swarms. For example, a developer using Robot Operating System (ROS) could create a `Variable Publisher` to collect data from ROS nodes. We implemented a `Variable Publisher` for ARGoS, supporting C++ and Buzz controllers. Within ARGoS, a loop function collects relevant data (i.e. robot variables) from the robots, running either C++ or Buzz controllers. A Transmission Control Protocol (TCP) server runs in parallel, taking collected data and broadcasting them to any connected clients. Initially we were going to use User Datagram Protocol (UDP) instead of TCP to make use of UDP broadcasting, but the TCP client proved easier to implement in Unity.

The ARGoS `Variable Publisher` was designed to be plug-and-play, allowing developers to simply drop the required files into their project and add the publisher loop function to their ARGoS configuration. Out of the box, it will look for any Buzz controllers and collect all global data from the controllers. C++ controllers need to extend the

`PublishableController` class, which enables the controller to self-report which data is to be published. A controller can add the value of a data stream by calling the `AddInt`, `AddFloat`, and `AddString` methods. These methods take in the name of the data stream and the corresponding value.

Deliverables

This project has many potential applications, and significant potential for future improvements. The scope of this project was gradually narrowed after conducting research, brainstorming features, and creating the preliminary design. We identified the four goals below as essential and focused our efforts on completing them. These goals were successfully completed over the course of this project.

Goal 1: Gather Data from Many Robots

From a design standpoint, this goal was the most critical to Dr. Swarm, and was a potential bottleneck for our application. Swarms can have large quantities of robots, leading to a large amount of information that needs to be sent from the robots to Dr. Swarm. The large amount of data provides network latency and leads to low network throughput. While TCP networks handle packet loss, it is negatively affected by latency. We resolved this by using a centralized controller and JSON protocol to send information. The centralized controller allows Dr. Swarm to receive information from one source instead of communicating with many separate servers.

For the set up in the NEST Lab, the application needs to be able to handle at least four sets of information from the robots. While this was not thoroughly tested, we ran several ARGoS experiments of collective transport and created multiple visualizations with the information received from the robots. As the experiments ran, there was no significant drop in performance based on the amount of information gathered or displayed from the four robots. We also ran a few tests where five robots could transmit information, and we had similar results with the application successfully running in real-time. The maximum number of visualizations the application can display at one time is five robots due to Vuforia limitations. With five robots being able to work, this goal was met.

Goal 2: Provide Intuitive Visualizations Conveying Information about Swarm Robots to Developers

We developed four types of visualizations and obtained feedback about the visualizations from several sources. Our research determined that line graph, pie chart, bar chart, and indicator visualizations would be the most successful in conveying information in real time. A line graph displays two sets of information in relation to each other. Generally, this is used with the x axis referring to time. Bar charts allow a developer to estimate current values and compare those estimates to each other. Pie charts are most useful for showing part-to-whole relationships. Indicators allow users to quickly display changing data and the state of the swarm.

Additionally, we consulted with swarm developers from the NEST Lab throughout development to gain iterative feedback about our visualizations. This enabled our visualizations to become more intuitive. We also iteratively received feedback from Professor Lane Harrison, an expert in data visualization. Through this iterative development, we have successfully completed this goal.

Goal 3: Easily Integrate with Existing Swarm Projects

There is no single set of software or single type of robot that every swarm developer uses, so we designed our application to be general purpose. This involves being able to track different types of robots as well as integrate with their software to send data to our application. In order to track different types of robots, a developer can create a target database with Vuforia. This database can then be imported into the Unity application and change the image targets on the game objects. This allows the application to track any type of robot in their swarm.

Utilizing a TCP server and client, our application can accept data from any type of robot, software, or programming language. A developer can write a `Variable Publisher` or use the one we provided for ARGoS, as described in the Implementation section. This enables the application to take information from ARGoS, ROS, or any other software. We tested this by creating two different publishers for ARGoS: a Buzz controller, and a C++ controller. Both were successful in sending out relevant data to the application.

Goal 4: Allow Compatibility with Different Devices Simultaneously

This goal has two parts: working with different types of devices (e.g. Android, iOS, AR headsets), and running simultaneously on multiple devices (e.g. the lab tablet and a staff's phone). Unity allows our application to easily port to Universal Windows Platform (UWP), Android, and iOS. This covers most widely available touch screen devices. Dr. Swarm is unable to port to standalone desktop apps due to limitations with Vuforia. Due to control scheme limitations of the user interface, the application only works with touch screen devices. We tested our application using both an iPad and an Android phone. Having the application run on multiple devices allows for multiple concurrent perspectives of the same swarm experiment. This allows multiple developers to observe unique data about the swarm. The TCP server and client enables this to happen with multiple devices.

Future Work

This project had significantly more potential than we could feasibly complete in 28 weeks. We chose to focus on implementing back end functionality, including essential visualizations and user interface features. This resulted in a strong framework for future projects to build upon. This section outlines the ideas, features and goals that we determined to be outside of our scope.

There were many features that would improve the visualization system in general. First, we had planned to create a 3D mesh visualization. This visualization would show how data changes relative to space. The magnitude of the data would be shown in the z-axis. A simple example of this would be visualizing the heat sensors of each robot in a swarm. If one area of the room was warmer than the rest, the visualization would show hills and valleys representing higher and lower temperatures, respectively.

Another major planned feature was the virtual monitor. Currently, visualizations are exclusively located above robots. We discussed allowing visualizations to exist elsewhere, detached from the robots. Users could create a virtual area which acts similar to a computer monitor, where a number of visualizations could be displayed at the same time. This virtual monitor could be fixed to a physical location or move around at the user’s behest. This would allow use of Dr. Swarm without AR as a dedicated diagnosing screen, as well provide a better location to display visualizations about multiple robots. Some additional features would need to be added to visualizations to make this workflow feasible, such as support for resizing visualizations.

There are also improvements to add to the visualizations themselves. On the front-end, the visualizations could use additional clarity, such as a title bar for each visualization that could show the name of the visualization. Some visualizations have keys which associate color to data being visualized, but others like indicators and line graphs lack this feature. Additionally, some parts of visualizations, such as axes and text, are difficult to see from far away. Tweaking of font and object sizes could make the visualizations more readable. Finally, our method to assign colors to graphical components needs to be reworked. Currently, only robots have a color associated with them. This was initially random but had to be set to a constant value due to technical limitations relating to Unity’s random implementation and threading. In order to associate a color to a data stream, we created a function that uses the Golden Ratio in order to assign a unique color to an undefined number of data streams. This worked quite well for our purposes, but there are better ways to use color for data visualization. We therefore identified multiple libraries for assigning color: the Tableau color pallet, ColorBrewer, and Colorgorical. One of these libraries would need to be chosen and implemented in the future. We also realized that the color of a data stream was not consistent across visualizations. A system needs to be implemented which associates a color to each data stream but still be individualized across the robots.

For the back-end, visualizations internally pass numerical values from the data class to the container class. This means that any textual data cannot be visualized. We realized that being able to show text above a robot would be a useful feature, which would require this back-end change. We encountered situations where restarting the application removes all visualizations, forcing us to manually recreate the same visualization that we were testing. Allowing visualizations to be specified through the network would solve this issue. In the JSON packet sent from the server, a new tag describing visualizations could be added. This way, a swarm developer could specify new types of packages, such as a manifest file that specifies the visualizations they want created when connecting the Dr. Swarm application to the swarm. These visualizations would be created immediately and would save time for users of Dr. Swarm.

The current user interface could use significant visual improvement. Additionally, there are still a few minor bugs and improvements to be made with the current functionality. This includes a better method of navigating between tabs and improving the editing process. Additionally, some screens with similar function and layout have inconsistent cosmetics.

The user interface uses simple raster graphics from the Unity game engine. Changing graphics makes sharper images with magnification occurring when using larger screen sizes. Potential improvements are switching to vector graphics or using libraries similar to d3 [11]. Other libraries can allow for the user interface to have more specification into what the visualizations look like and function.

To make the visualizations and UI better, there are several potential improvements to the AR aspects of the application. Adding mixed reality (MR) to Dr. Swarm would expand the capabilities of the application. This could allow for the addition of a virtual robot into the swarm as a control or for it to cause errors. It can also make the visualizations interactive and make it easier to interact with the physical robots. This would provide an immersive tool, allowing diagnostic capabilities beyond simple visualizations.

Our initial plan was to implement Dr. Swarm on AR headsets, such as the Hololens [12] or Magic Leap [13]. However, this was financially unfeasible. The AR headset would give the user a wider field of view and enable better tracking of the robots. Using an AR headset would require a compatible user interface. An Xbox controller, or similar hardware, could prove useful for controlling UI elements.

The limitations of Vuforia are acceptable for the NEST Lab and our preliminary efforts, but are inconvenient for a general-purpose swarm tool. Changing the AR software would make it easier to track more than five objects. This could also resolve other issues, such as being able to increase distance between the robots and the user, targeting smaller objects, and targeting 3D objects. A new AR software could mean that Dr. Swarm could work with Kilobots.

Our application lacks any formal testing. An in-depth user study that evaluates the usability and performance of the application could identify specific strengths and weaknesses of Dr. Swarm. This is most important in ensuring clarity of necessary and relevant information for the application. Additionally, this would prove the significance and scope of the application's uses.

Finally, the Dr. Swarm application could be integrated with other tools to create a full-featured swarm development tool. The NEST Lab has created an application to control swarms in AR and is planning on creating a debugging tool that works in AR as well. Integrating these three tools together could create a very powerful general-purpose swarm development application. A suite of tools like this has never been created for swarm robotics. To make this work, the networking portion of Dr. Swarm would need to be expanded to receive other types of information from the swarm. The networking would likely be a subsystem common to all three tools, so it is likely that the network code would be rewritten altogether.

Conclusions

This project was developed to expose the behavior of swarm systems through intuitive visualizations and assist with troubleshooting swarm applications. A general-purpose diagnostic tool outside of Dr. Swarm does not exist, making swarm diagnostics a daunting task for swarm developers. To combat this, we created an application that provides four types of AR visualizations that update with real-time information from the swarm. Developers can take information from the application and use it as a starting point to look at the system, beginning the diagnostic process.

Throughout our project we learned three key lessons. First, AR allows a unique opportunity to see relevant information in real-time by overlaying images in the physical world. This statement is broad, but true to our project. Throughout development, we experienced the uses and benefits of seeing swarm information in real-time. Second, we learned that reactive programming enables highly extensible design. This allowed our application to seamlessly update in real-time without creating latency in the application. Finally, we learned that visually representing data makes swarm robotics more accessible, especially to non-experts. We are not experts about swarm robotics, but by creating and using our application, we learned more about how swarm robotics works, and how exciting the technology is.

References

[1] E. Şahin, “Swarm Robotics: From Sources of Inspiration to Domains of Application,” in Swarm Robotics, 2004, pp. 10–20.

[2] T. Yasuda and K. Ohkura, “Collective Behavior Acquisition of Real Robotic Swarms Using Deep Reinforcement Learning,” in 2018 Second IEEE International Conference on Robotic Computing (IRC), 2018, pp. 179–180.

[3] M. Gauci, M. E. Ortiz, M. Rubenstein, and R. Nagpal, “Error Cascades in Collective Behavior: A Case Study of the Gradient Algorithm on 1000 Physical Agents,” in Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, 2017, pp. 1404–1412.

[4] F. Ghiringhelli, J. Guzzi, G. A. D. Caro, V. Caglioti, L. M. Gambardella, and A. Giusti, “Interactive Augmented Reality for understanding and analyzing multi-robot systems,” in 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2014, pp. 1195–1201.

[5] A. G. Millard et al., “ARDebug: An Augmented Reality Tool for Analysing and Debugging Swarm Robotic Systems,” Frontiers in Robotics and AI, vol. 5, Jul. 2018.

[6] Dey, Arindam, Mark Billinghurst, Robert W. Lindeman, and J. Edward Swan, ‘A Systematic Review of 10 Years of Augmented Reality Usability Studies: 2005 to 2014’, Frontiers in Robotics and AI, 5 (2018) <https://doi.org/10.3389/frobt.2018.00037>

[7] W. Hoenig, C. Milanes, L. Scaria, T. Phan, M. Bolas, and N. Ayanian, “Mixed reality for robotics,” in Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on, 2015, pp. 5382–5387.

[8] Open Source Robotics Foundation (n.d.). ROS.org | Powering the world's robots. [online] Ros.org. Available at: http://www.ros.org/ [Accessed 24 Apr. 2019].

[9] B. S. Le, V.-L. Dang, and T.-T. Bui, “Swarm Robotics Simulation Using Unity,” Faculty of Electronics and Telecommunications, University of Science, VNU-HCM, Vietnam, 2014.

[10] C. Pinciroli, V. Trianni, R. O’Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. D. Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo, “ARGoS: a

modular, parallel, multi-engine simulator for multi-robot systems," Swarm Intelligence, vol. 6, no. 4, pp. 271–295, 2012.

[11] Bostock, M. (2019). D3.js - Data-Driven Documents. [online] D3js.org. Available at: https://d3js.org/ [Accessed 24 Apr. 2019].

[12] Microsoft (2019). Microsoft HoloLens | Mixed Reality Technology for Business. [online] Microsoft.com. Available at: https://www.microsoft.com/en-us/hololens [Accessed 24 Apr. 2019].

[13] Magic Leap Inc (2019). Magic Leap. [online] Magicleap.com. Available at: https://www.magicleap.com/ [Accessed 24 Apr. 2019].