# Remotely Controlled Video Surveillance Monitoring System Using Microsoft Robotics Studio

by

Saeed Nikbakht

Submitted in partial fulfilment of the requirements
for the degree of Master of Applied Science

at

Dalhousie University
Halifax, Nova Scotia
December 2014

*To my dear parents, Mohsen and Shahla*

# TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# ABSTRACT

Robots consist of several mechanical and electronics parts. To accomplish a task, concurrency and coordination between these parts are required. The complexity of this combination makes programming of a robot complicated and time consuming. Robotic development environments (RDEs) were introduced to address this issue by providing a framework for robotics developers, which consists of programming tools, program components, and solutions to common robotics challenges. Robotic programmers can take advantage of the tools shipped with an RDE and use already developed components in their programs. These robotic frameworks offer some solutions such as teleoperation, distribution of processes, and predictive display. Some of them also provide a simulation environment where developers can test their codes on a simulated model of a robot, without creating or purchasing the actual robot.

In this thesis, some of the most popular RDEs are briefly reviewed and one is selected. Microsoft Robotics Development Studio (MRDS) is the RDE that was chosen to program an existing wire-suspended video monitoring surveillance robotic system. The service-oriented architecture of MRDS makes all programs developed in this framework capable of working with the network and using the Internet as an infrastructure for teleoperation. Also, the same architecture allows distribution of processes (services) between multiple computers. In addition, the powerful simulation environment of MRDS could be used as a predictive display, allowing operators to predict and monitor the reaction to their commands; when the simulated model of the robot responds to the operator's commands at the same time as the actual robot. The goal of this thesis was to validate several solutions of MRDS and examine its ease of use and its helpfulness in speeding up the implementation of robotics programs.

Despite the capabilities these RDEs provide, they present unique challenges for programmers. This experiment was completed by developing new MRDS services, using the existing components, and customizing the existing codes. Some of the features of MRDS are demonstrated in the final project, letting the operator control the robot over the Internet simultaneously with the simulated model of the robot. Also, distribution of processes between several computers is tested successfully. The experiment showed that the selected RDE has powerful capabilities and helpful tools for developers but it has a long learning curve due to its complexity which also slows down the development of services and hence the delivery of the final product.

## LIST OF ABBREVIATIONS USED

| | |
|---|---|
| 2D | Two Dimensional |
| 3D | Three Dimensional |
| BAM | Bluetooth Adaptor Module |
| BSD | Berkeley Software Distribution |
| CCR | Concurrency and Coordination Runtime |
| DLL | Dynamic Link Library |
| DSS | Decentralized Software Services |
| GDD | Generic Differential Drive |
| MRDS | Microsoft Robotics Development Studio |
| NMC | Network Modular Control |
| PID | Proportional Integral Derivative |
| RC | Radio Controlled |
| RDE | Robotic Development Environment |
| VPL | Visual Programming Language |
| VSE | Visual Simulation Environment |

## ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Dr. Ken Wilkie, for the encouragements, his assistance and guidance, useful comments, and for providing all resources and materials through the learning process of this master thesis. I thank Dr. Watts, Dr. Gu and Dr. Kerhmanshahi-Pour for their interest and willingness to be part of my guidance committee.

Furthermore, I would like to thank my brother-in-law, Dr. Ramin Sadeghi, for setting me on the right path to finish my thesis and my wife, Mehrafrooz, for her support.

# CHAPTER 1

# INTRODUCTION

## 1.1 PURPOSE OF THIS STUDY

Robotics is finding its way more than ever into all aspects of our lives. There are more innovations and research happening in this field as people find an increased number of uses for robots in different areas. From self-driving cars (Google is to start building its own self-driving cars, 2014) to pizza delivery drones (Russian pizza chain claims to deliver by drone, 2014) to placement of robot avatars for absent fans in stadiums (Korean baseball team fills stands with 'Fanbots', 2014), every day there is something new about how robots are being used. To keep up with the fast paced innovations, it is important for robotics researchers to be able to quickly and inexpensively prototype their new ideas. Development of a robotics system is a multidisciplinary task. A functional robot consists of mechanical structures, electronics parts and circuits, computer programming, and maybe logic for artificial intelligence. It is important to be able to design the system's prototype quickly and cheaply (Zoppi, 2012). One of the most time consuming processes in robotics development is programming of the robot, often from scratch. In order to reduce the development time and cost, the solution is to set a standard and make reusable modules so developers can concentrate on adding new modules rather than recreating the old ones.

The objective of this study is to develop the software and control systems to demonstrate remote control of a robotic surveillance system using one of the new solutions for robotics development. The robotic system used in this study is a wire-suspended video surveillance system developed by Dr. Ken Wilkie. The original controlling program of the robot was developed in LabWindows/CVI, a language based on ANSI C and distributed by National Instruments. Robotics Development Environments (RDEs) appeared to be a suitable approach by providing standards and ready to use features for rapid development of robots. The goal was to redevelop the program using an RDE in such a way that it would make the

robot accessible via the Internet, allowing operators to remotely control the robot with an Internet connection.

## 1.2 Significance Of This Study

This work highlights some of the pros and cons of an RDE during a practical exercise of the implementation of a custom robot. After some research, an RDE developed by Microsoft, Microsoft Robotics Development Studio (MRDS), was selected. This development environment was new, at the start of this thesis, and it was based on C# language, a popular and powerful language with support from Microsoft. The initial tutorials and articles published by Microsoft looked promising and it seemed to be the right choice of an RDE for the goals of this study. MRDS is a service-oriented development environment so by design the processes are accessible through the network. The simulation environment can simulate the real world as the physics rules are applied to it and has ability to be associated to a service, letting the simulated model to be controlled the same way the actual robot is controlled.

## 1.3 Approach

In order to become more familiar with the framework, first a test system was programmed using the 'out of the box' Microsoft Robotics Development Studio (MRDS) services. After a successful test, the main project was started. Step by step, new services and 3D simulated models were implemented and tested to get the whole system ready and executable in MRDS. Development of the program to control the custom robot required several features and abilities of MRDS to be tested.

## 1.4 Thesis Organization

The background of the Robotic Development Environments (RDEs) and their impact on robotics are reviewed in Chapter 2. The chapter continues by reviewing some popular

robotics software platforms, and previous works completed using them are compared. In Chapter 3, a demonstration project in MRDS is presented and the results of the testing are discussed. A pan-tilt camera is implemented in Chapter 4 and its architecture and services are explained in detail. In Chapter 5, the implementation of hardware of the actual robotic surveillance system as well as its simulating model, and how it is manipulated by MRDS are discussed. The study's conclusions are given in Chapter 6 where recommendations for future work are also discussed.

# CHAPTER 2

# BACKGROUND

## 2.1 ROBOTIC DEVELOPMENT ENVIRONMENT

Robotics is a field whose time has not quite arrived, but it is expected that dramatic growth in this field will soon occur. The current state of robotics has been compared to that of the PC industry 30 years ago (Gates, 2007). Due to its relative immaturity, the field of robotics still faces many challenges. One of these is the lack of standards for both hardware and software. This lack of standards has led robotics researchers to recognize the need to develop what Kramer and Scheutz (2007) call Robotic Development Environments or RDEs.

RDEs consist of ready to use components that minimize the need of coding. Among these features there is a simulation environment that allows developers to design and test the robot before creating the physical prototype. Robots used to be shipped with their control software mostly locked in by the manufacturer. The licensed software made it impossible to change the code and it has always been a challenge for developers to change or extend these software applications as they usually required access to the code and low-level programming. Other challenges which have slowed down improvement in the robotics industry are inflexibility, complexity and lack of portability, scalability, and extendibility of robot software applications. Software applications developed for specific hardware systems could not be reused in other robots. Since these software applications were mostly not object-oriented and were not developed as independent components, any changes requires modification in the original code which, often without good documentation, could further increase the complexity of code. The old robotic software systems were also missing support for teleoperation, multi-robot application and distributed tasks.

Over time, developers and researchers in robotics looked for a solution. They needed a software which could be reusable on different robots and hardware, speed up the development by letting developers use already-developed components, and does not require

them to develop software programs from scratch for every new robot. The result was a framework, also referred to as RDE.

## 2.2 POPULAR ROBOTICS FRAMEWORKS

There are several companies and research teams working on robotics frameworks, but not all of them are active or accepted by the community. Some of the most popular frameworks are Robot Operating System (ROS) by Willow Garage; Player/Stage/Gazebo, which is an open-source project developed at University of Southern California; LEGO Mindstorms by LEGO; and Microsoft Robotics Development Studio (MRDS) by Microsoft.

In this section, the above robotics frameworks will be first briefly overviewed before reviewing some of the studies that have compared them.

### 2.2.1 ROS

Published first in 2007, it is a BSD (Berkeley Software Distribution) open-source license robotics framework that can run on UNIX, OS X and Windows operating systems (ROS Tutorials, 2013). ROS could gradually attract many developers in robotics and it now has an active community.

Similar to operating systems, ROS provides message passing between modules. By defining a standard format of messages and leaving serialization and deserialization of messages to ROS, it can support communications of modules developed in different languages. ROS libraries are available in several languages: C++, Python, Java, LISP, Octave, and Lua (Quigley, et al., 2009).

Nodes, messages, topics, and services are the main principles of ROS. In ROS, nodes represent processes, e.g. camera node is a module that could process all visual data, and topics are what nodes communicate based on them. A node can publish a topic and another node can subscribe to that topic to receive messages sent by the other node. Services are another way for nodes to communicate by sending requests and receiving responses (Quigley, et al., 2009).

### 2.2.2 Player/Stage/Gazebo

**Player** is a free open source software application that provides tools for controlling single or multiple robots. It first became available to the public on the Interaction Lab web site in 2000. Player can work with any type of controller over a network as long as the controller can communicate through a TCP socket.

Player was designed to be language and platform independent and it has tools for different robots and devices. Player has APIs in different languages and the community contributes to it by expanding a number of supported languages. The Player server can run on Windows, Linux and Mac OSX (Biggs, Rusu, Collett, Brian, & Vaughan, 2012).

**Stage** is the 2D simulator which became available with Player before Gazebo, the 3D simulator, became available. Stage is designed to simulate a high population of robots with low fidelity. It simulates robot platforms and sensors and allows hundreds or thousands of robots to be simulated at the same time. It provides collision avoidance and map generation and can run on Linux and Mac OSX (Gerkey, Vaughan, & Howard, 2003).

**Gazebo** is the 3D simulator of this robotics framework. It is compatible with Stage programs allowing all code execution with no change. Gazebo allows developers to create 3D environment as well as 3D robots. All objects have mass, friction, and other attributes which allow their simulation be closer to reality. Gazebo is designed to simulate a low population of robots with high fidelity (Koenig & Howard, 2004).

### 2.2.3 Lego Mindstorms

It is a graphical programming language created based on National Instrument's LabView program. There are blocks for reading sensors, controlling motors and programming is done by dragging blocks and connecting them to each other. It was created for children so simplicity was one of the goals which results in limitation (Griffin, 2010).

### 2.2.4 MRDS

It is a robotics framework by Microsoft that was released in December 2006 to address challenges in robotics programming by introducing new technologies for managing

concurrent operations and distributed processes. It was designed to facilitate communications between decoupled software modules and hardware components. MRDS provides a powerful 3D simulator and simple programming method called Visual Programming Language (VPL) and similar to most of Microsoft products it only runs on the Windows operating system. The framework is developed based on .Net and it supports C#, Visual Basic, Managed C++, and IronPython programming languages (Jackson, 2007).

A systematic review conducted by University of São Paulo (Oliveira, L. B. and Osório, F. S. and Nakagawa, E. Y., 2012) studied some of the robotics system tools based on Service-Oriented Architecture (SOA). In their study, MRDS was found as the most cited service-oriented robotics development system. However, the authors of the review believed that the popularity of ROS has started to rise and that it will more than likely become the most popular.

Another study done on commercial and open-source robotics simulator software systems was conducted by Staranowicz and Mariottini (2011). Some of the characteristics of these simulators are listed in a table making it easier to compare them (Table 2-1).

Parameters used in their research were (1) Operating System: which OS is supported by the software system, (2) Simulator type: whether it is a 3D or 2D environment, (3) Programming language (4) Documentation: level of documentation ready by the software (5) Tutorial: whether examples and step-by-step instruction is provided or not. Limited means help exist but it is not enough. (6) Portability: if yes then the simulation code is separated from the platform tool (7) Sensor: list of supported sensors between most requested ones. (8) Debugging/Logging (9) Graphical User Interface: describes if it is possible to change objects and the environment during run-time and/or program functions in a development environment.

Table 2-1. Characteristics of Open-Source Robotic Simulation Software Systems (Staranowicz & Mariottini, 2011)

| | Player/Stage | Gazebo | ROS | Simbad | CARMEN | USARSim | MRDS | MissionLab |
|---|---|---|---|---|---|---|---|---|
| **OS** | Linux, Mac, Win. | Linux | Linux, Mac, Win. | Linux, Mac, Win. | Linux | Linux Win. | Win. | Linux |
| **Simulator Type** | 2-D | 3-D | 2-D, 3-D | 3-D | 2-D | 3-D | 3-D | 3-D |
| **Programming Language** | Player(any) Stage(C, C++, Python, Java) | C, C++, Python, Java | C++, Python, Octave, LISP, Java, Lua | Java | C, Java | C, C++, Java | VPL, C#, Visual Basic, JScript, Iron-Python | VPL |
| **Documentation** | Low-Level | Low-Level | High-level | High-Level | Low-Level | High-Level | High-Level | High-Level |
| **Tutorial** | Yes | Yes | Yes | Limited | Limited | Limited | Yes | Limited |
| **Portability** | Yes | Yes | Yes | Limited | Yes | Yes (using Player) | Yes | Yes |
| **Sensors** | odometry, range | odometry, range, camera | odometry, camera, range | vision, range, contact | odometry, range, GPS | odometry, range, camera, touch | odometry, range, camera | odometry, range |
| **Debugging/ Logging** | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes |
| **Graphical User Interface** | No | No | No | No | No | Yes | Yes | Yes |

## 2.3 SELECTING THE RIGHT FRAMEWORK

When the first research on finding the right development tool was done for this project in 2007, Microsoft recently introduced its new robotics framework Microsoft Robotics Development Studio (MRDS). At the time there was no book published on this topic and the main source of information was mostly the tutorials and videos provided by Microsoft.

Based on the available information, the future of this framework seemed bright especially as it was created by a major software company with years of experience in development of enterprise applications. The new technologies used as the core of this framework were powerful and intelligent solutions for addressing robotics challenges. The framework was also delivered with a powerful simulator that benefited from a powerful simulation engine (PhysX) which is used in major 3D video games. Another parameter which made this framework very attractive was the fact that it has been offered free so the budget was not a show stopper to start using it.

Microsoft attracted many famous large companies and worked with them to support their products in MRDS. The services and simulated models of some of these robots like iRobot Create, Lego NXT, and Kuka articulated arm are included in the framework. The author found working with out-of-the-box standard robots was initially straight forward; however challenges increased as the project continued and he tried to use the framework for developing a custom robot.

## 2.4 MICROSOFT ROBOTICS FRAMEWORK

MRDS was first introduced in 2006. The application got more attention after publication of an article by Bill Gates in Scientific American magazine (Gates, 2007), where he explained his vision on the future of robotics and its growing use in the home. The article explained that MRDS was designed in response to the demand of a framework tool and standardization of robotic programming to facilitate faster improvement in this field. Gates stated that the time was similar to the 70's when many companies were developing their own Personal Computers and Operating Systems until Windows OS was introduced and dominated the world.

Since its release, MRDS has been used in several applications including non-robotics applications. ABB, a leading supplier of industrial robots and robotics software, is one of the first companies that took advantage of MRDS. ABB provided a connector, ABB Connect, which allows students to design and implement virtual robots and build controllers for them (Microsoft, 2008). Some other examples of the commercial use of this framework are robuBox (D. Salle, 2007) by Robosoft, which "is a software for the design of generic robotics controllers implementing advanced robotics functions and behaviors", and the MySpace site which used the main component of MRDS, CCR, for a non-robotic purpose solution to improve functionality of their web application. Another example is SAFAR (Software Architecture For Agricultural Robots), a software for agricultural robot simulation which was developed by combining MRDS with Google Earth (SAFAR, 2012).

Like Microsoft's experience with Windows, the similar marketing strategy might help to guarantee the future of MRDS by making it a profitable product for the company. In an article published in 2007 in IEEE Spectrum magazine (Cherry, 2007), the author explains about Microsoft's plan to make profit on this framework by charging the users for licensing

copies of this software however that strategy has been halted after release of MRDS 2008 and so far the new versions are accessible free of charge. Although in the same article the author talks about the licensing charge in future commercial robots, which are built based on MRDS; something similar to Microsoft license fee for PCs sold with the Windows OS. He concluded that one could predict more support from Microsoft in this field and possibly more growth in improvement of this framework.

### 2.4.1 Core Features

Microsoft robotics framework is powered by two very advanced technologies, Concurrency and Coordination Runtime (CCR) and Decentralized Software Services (DSS). These .NET libraries were first released with MRDS toolkit but due to their usefulness in other applications they were later issued separately and since then they have been used in several commercial and academic applications.

In robotics programming managing of I/O communications with sensors and actuators concurrently and at the same time being responsive to events in shortest time are essentials. Threading has been used for this type of system but it could cause a lot of complexity due to the nature of this method of programming. Known issues such as deadlock, race and handling errors are some of the challenges developers face in thread programming.

Concurrency and Coordination Runtime (CCR) library was first designed to take advantage of multicore multiprocessor systems but it was later considered as a solution for robotics programming (Gates, 2007). CCR focuses on solutions for concurrency, coordination of asynchronous applications to simplify complexity of such programs and improve the performance of applications (Johns & Taylor, 2008). Using this solution requires a new way of programming different to traditional threading programs. In CCR processes can run parallel and communicate the results with each other. This is made possible by messaging which is the basis of programming with CCR.

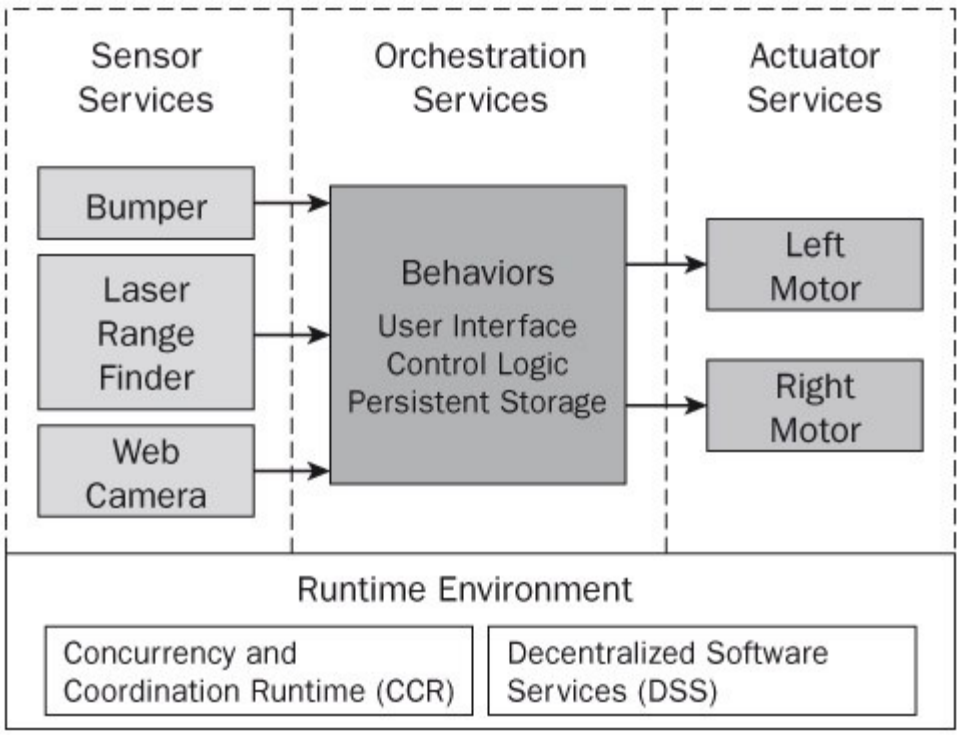


Figure 2.1. Relation of CCR and DSS with services in an example of a robot implementation. DSS and CCR are shown in the same level as services can make direct calls to CCR (Johns & Taylor, 2008).

Decentralized Software Services (DSS) adds another layer to CCR. It lets CCR multiple task processes run as services. Services are processes enabled to receive messages (aka requests) and send structured data in response over the network. Each service has a state associated with it and this state can be changed based on the type of messages the service receives. The service can also send additional messages or notifications to other services. Services can subscribe to other services to be notified once the state of a service changes. Services can also partner with other services to send message and receive responses from them (Johns & Taylor, 2008). DSS also provides distribution of these services on different machines so no matter what machine hosts a service it can communicate with other services via the network. DSS is designed based on REST model which makes services available to other applications or services and as result low coupling become possible in MRDS.

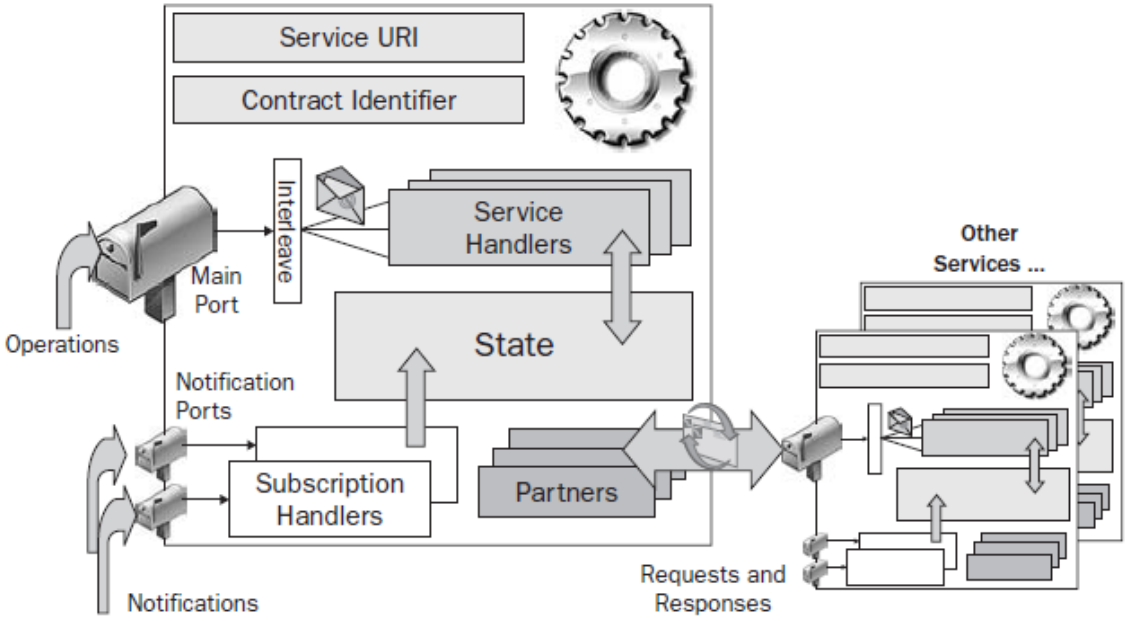


Figure 2.2. All components of services in DSS (Johns & Taylor, 2008)

DSS allows processes to be taken out of the robot and run on PCs. Therefore robots can be built as inexpensive devices that wirelessly communicate with PCs that provide all the intelligence required for the robot. DSS also make this possible for human operators to connect to the robot over the network using Web Applications.

### 2.5 Simulator and VPL

Microsoft Robotics Developer Studio includes a simulation environment using a physics engine developed by Ageia (Johns & Taylor, 2008). The tool comes with some indoor and outdoor scenes. It also includes several simulated robots such as LEGO NXT, iRobot Create and KUKA LBR3 robotics arm. The editor of the simulator allows users to prototype custom robots and to add objects like doors, walls, and furniture to the scenes. This tool allows developers to simulate their robots and debug their algorithms in the simulated environment before creating the real robot.


Figure 2.3. A simulated scene in edit mode and list of entities on left side menu (Microsoft, 2012)

Simulation scenes are built of entities. Entities are objects with mass, size, and other physical attributes. An entity can be as simple as a box or a sphere. Entities can have a mesh associated with them that make them look more realistic. The simulation engine applies forces like gravity and friction to entities and it can detect collisions between entities.

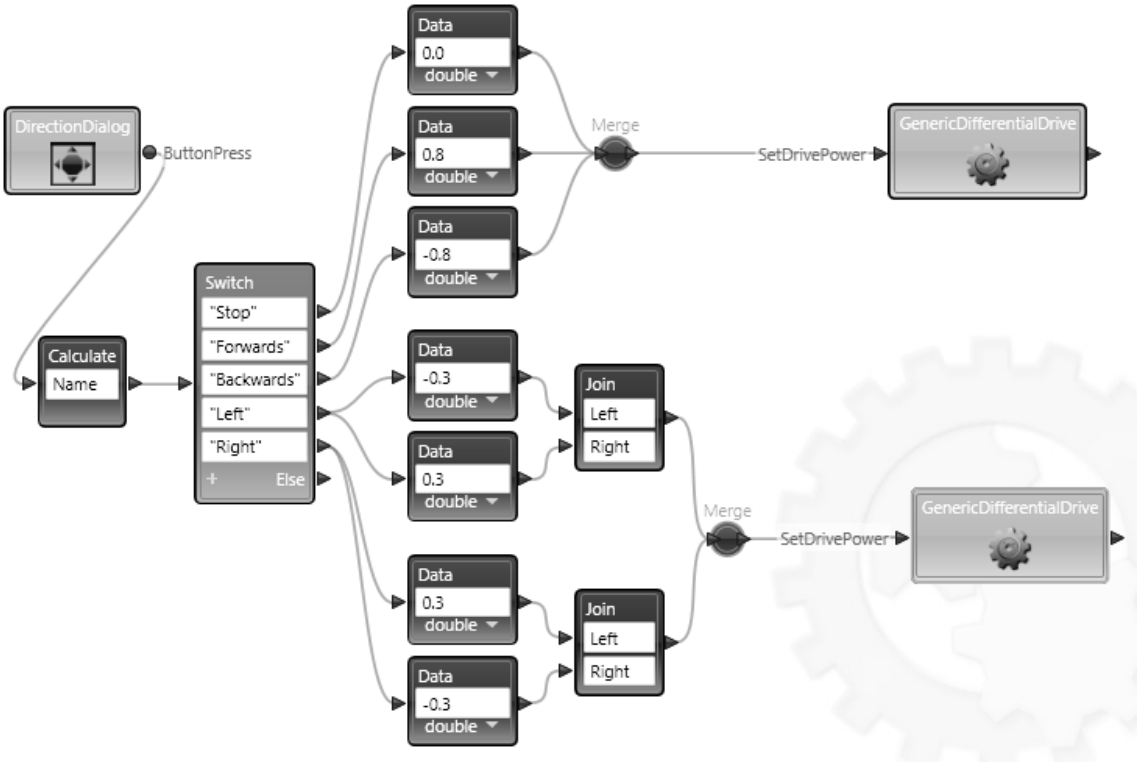


Figure 2.4. Example of a VPL program (Microsoft, 2012)

The other component introduced with this toolkit was Visual Programming Language (VPL). It is a solution for developing scenarios without coding. This graphical language, that is included in MRDS, allows developers to create orchestration services, which are a combination of low level services, by defining how data is transferred between services. Still coding is required for low level services that mostly interface directly with hardware but for the orchestration service it is sometimes possible to use only graphical diagrams in VPL (Figure 2.4).

# CHAPTER 3

# THE PILOT PROJECT

The idea behind this preparation phase was to get hands-on experience with MRDS on an experimental setup similar to the final project but on a smaller scale. The simulated robot and all services used in this experiment were out of the box services delivered by MRDS. This would help to get a better understanding about how the main project could be accomplished in a shorter period of time. In addition, some of the services and entities used in this phase, for instance the webcam service and the gamepad service, were later used in the final project.

## 3.1 SOFTWARE IMPLEMENTATION

The demonstration project was started by creation of the simulation scene. Since the simulation of the selected robot, iRobot Create, was already included in MRDS, the first step was to create a simulated scene, add an entity of the robot to the scene and then to write a simple VPL program to drive it around. The idea was to make the scene an exact replica of where the actual robot was going to be tested. An area in Dalhousie Biorobotics Lab was selected for this experiment. The area was measured and the Maze Simulator was used to create the simulated scene. The Maze Simulator is a community-developed MRDS service, which allows the user to create a scene of different walls using a bitmap file. Although this tool helps to develop a scene faster, it is able to add only a few types of basic entities to the simulated scene. In order to make the created scene more realistic, additional edits were required. Door entities (shown in green) and the virtual wall entity (shown in red), and wall and floor textures were later added to the scene using Microsoft Visual Simulation Environment (VSE).

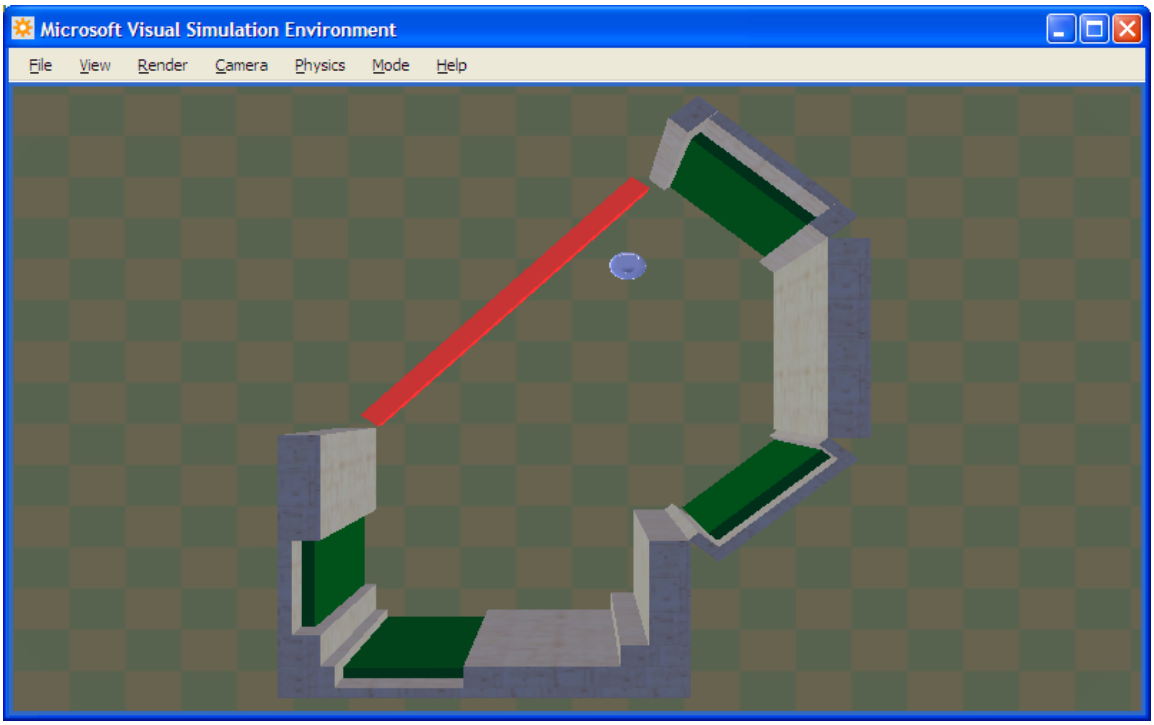


Figure 3.1. Top view of the simulated scene showing the iRobot Create as a gray circle

VPL was selected for programming because it was less complex and programming by this language was quicker. Since this experiment did not need development of any new services, VPL could cover the requirements for the development of this pilot project. In VPL, there is a generic contract (e.g. abstract service) for Differential Drive that specifies a programming interface for driving a robot regardless of its type. The implementation of the drive service for a specific robot can then be assigned to the service by changing its manifest (a list of all the required services and their partnerships). The Create is a two-wheeled robot and like many other mobile robots, it has a differential drive which lets each of its wheels be driven independently. By powering each wheel properly, the robot can move forward and backward, turn left and right, or rotate on a spot without changing its position. In this experiment, a gamepad was used to control the robot. As the gamepad axes change, X and Y axes of the gamepad are set to an integer number between -1000 to 1000.

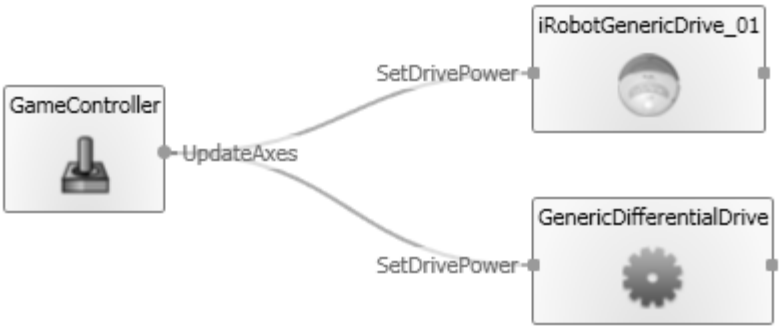


Figure 3.2. UpdateAxes notifications sent to Generic Differential Drive services set the power of wheels.

To drive the simulated robot, a Generic Differential Drive (GDD) service was used. The manifest of the simulated scene was then associated to the GDD service so when it is executed the service is automatically assigned to the differential drive of the simulated Create in the scene. For the actual Create robot there is already a service in VPL called iRobot Generic Drive. The manifest of this service was not changed as it is already set to the right manifest by default. To control both simulated and actual robots simultaneously the same Game Controller service was partnered with both drive services. In the VPL program, by connecting the `GameController` service block to the GDD block, GDD service subscribes to `GameController` service to receive notifications sent by this service. As illustrated in the following figure (Figure 3.3), `UpdateAxes` notification triggers `SetDrivePower` operation on both drive services. GDD services expect a number between -1 and +1 for setting the power of the wheels. Therefore, the received values of X and Y axes are converted in the following equations before they are sent to set the power of each wheel:

LeftWheelPower = (-Y + X) / 1000

RightWheelPower = (-Y - X) / 1000

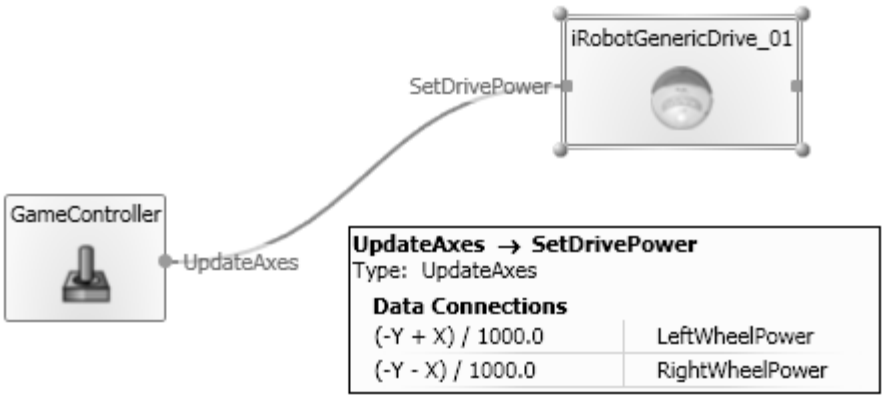


Figure 3.3. Data connection of the messaging connection between GameController and one of the GDD services

For the actual robot programming, more VPL blocks were required to let the operator control the robot via the GUI controller (`DirectionDialog` Service) in addition to the gamepad. This option was added to let the operator move the actual robot without moving the simulated robot. As a result, the actual robot's position could be adjusted to the simulated robot's position before starting the synchronized drive test.

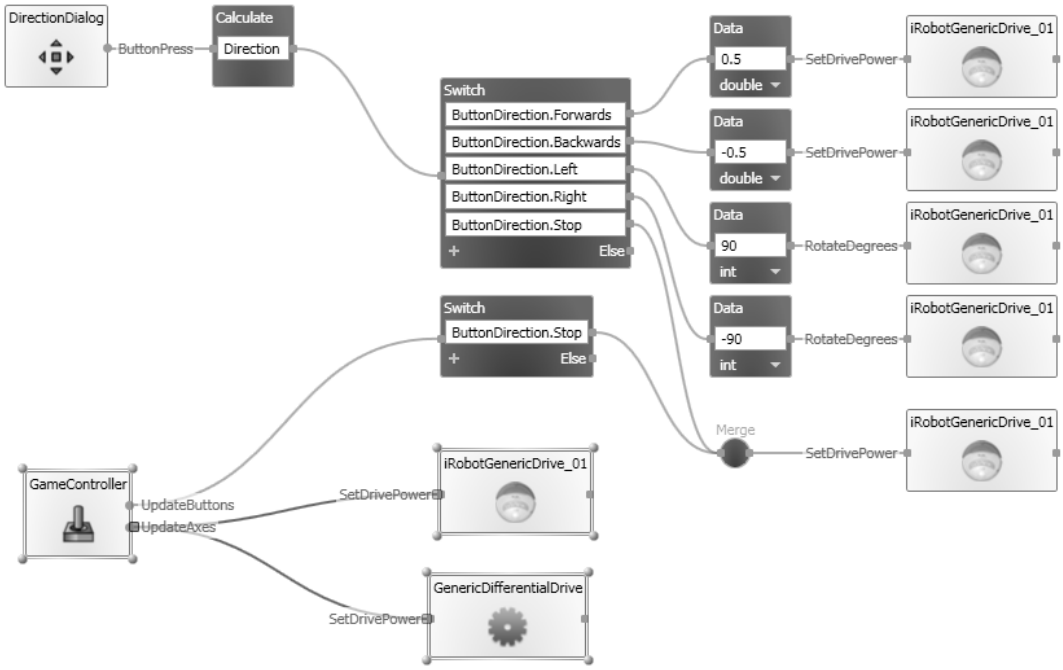


Figure 3.4. The VPL program running on the Operator PC

The above figure (Figure 3.4) illustrates the complete VPL program that ran on the remote computer used by the operator (Operator PC). The drive service of the Create robot, `iRobotCreateDrive`, is only added to define its partnership with the `GameController` service but the service itself was not hosted on the Operator PC. In the manifest file of this VPL, the URL of `iRobotCreateDrive` service was manually changed to point to the IP address of the computer located on the site where the robot is located (Onsite PC). This is the computer that hosted `iRobotCreateDrive` service (Figure 3.5).

On the Onsite PC the following VPL was executed to launch the Webcam and iRobotCreateDrive services on the PC.


Figure 3.5. The VPL program running on the Onsite PC

## 3.2 HARDWARE IMPLEMENTATION

### 3.2.1 iRobot Create Robot

As mentioned above, iRobot Create was selected as the hardware platform for this experiment. This robot was first introduced in 2007 after iRobot realized the interest of robotics researchers in its commercial robot, Roomba. The Create is a programmable mobile robot with two motor driven wheels, left and right bumper sensors, and edge detectors. This platform is inexpensive, durable, and extendible.

This robot is one of the supported robots by MRDS. So the simulated model and its services are already included in the toolkit. Among other MRDS supported mobile robots, the Create was selected because its strong platform is steady enough to support the webcam and the wireless transmitter required for this experiment. The cost constraints of this project were also a concern. Weiss and Overcast studied a couple of popular robots to be used for educational purpose (2008). In this study they concluded that the Create is a good choice where durability and affordability are the main factors.

Two components were mounted to the robot, one for communication with the robot and one for streaming out videos. The first component, the Bluetooth Adaptor Module (BAM), was a Create accessory used for communication between computers and the robot. BAM connects to the serial port of Create and allows applications running on a computer to wirelessly communicate with the robot through a Bluetooth connection. The second component was a standard analog surveillance camera connected to an Audio/Video radio transmitter. The transmitter converts the Audio/Video data to 2.4 GHz radio signals. For streaming video there was an alternative of using an IP camera; however, this camera was very expensive at the time of this experiment, and it was decided to use a cheaper solution to stay on budget.

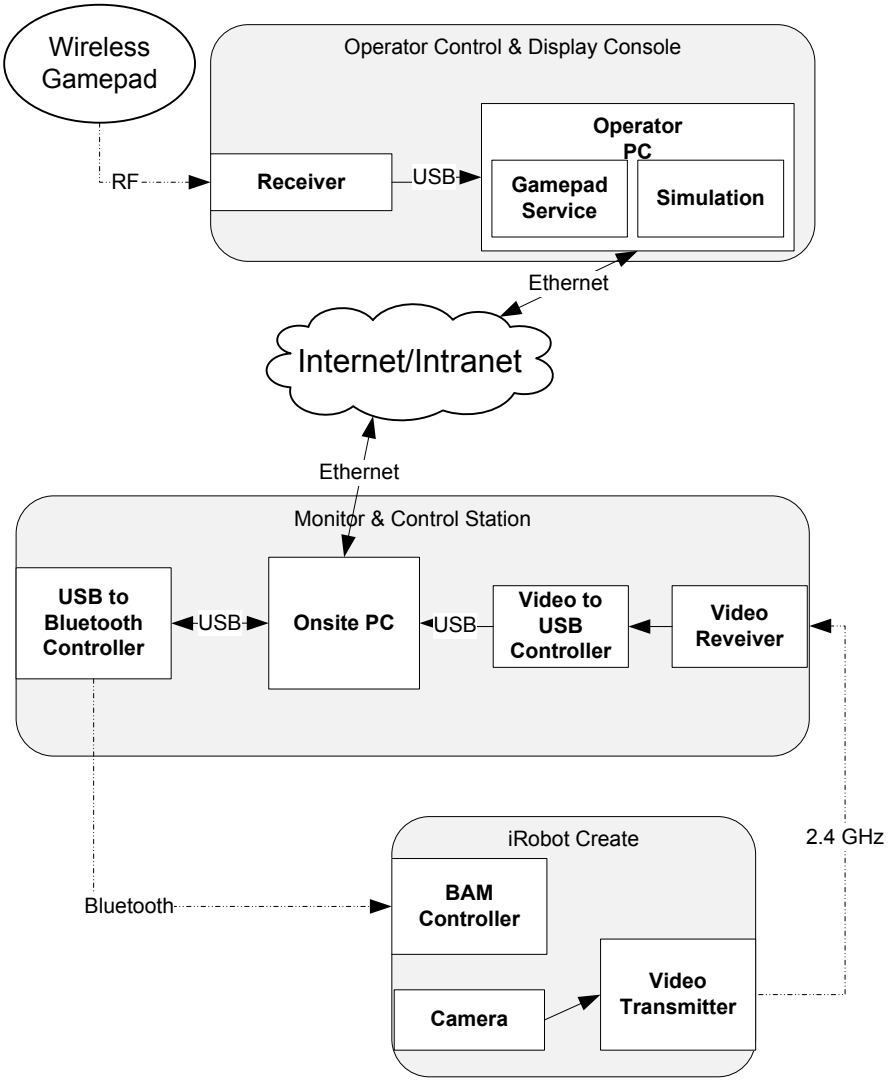


Figure 3.6. Hardware implementation of the pilot project

### 3.2.2 Onsite PC

The Onsite PC was the computer located in the same location as the robot. It was used to operate and communicate with the robot wirelessly. This computer received video streams from the robot. This PC was a Windows XP computer with MRDS running on it hosting iRobot Create Differential Drive service (for driving the robot) and Webcam service (for streaming videos). The PC did not have embedded Bluetooth so a Bluetooth USB adaptor was hooked to it to let it communicate with the robot. Also a 2.4 GHz wireless Audio/Video receiver was used to receive streams from the camera on the robot. This

receiver was connected to an Audio/Video adaptor to convert analog outputs of the receiver to digital and send them to the computer through the USB port.

### 3.2.3 Operator PC

The operator PC was the PC used by the operator to send commands to the robot. The PC was connected to the Onsite PC by the Internet and commands received from a wireless gamepad were sent to the Differential Drive service running on the Onsite PC. The operator was receiving video streams from the Onsite PC and at the same time could match them to the streams of the simulated environment. MRDS was also running on the Operator PC to host the simulated environment and the gamepad service.

## 3.3 DEMONSTRATION SCENARIO

The scenario was to drive a mobile robot with a mounted camera in an area in Dalhousie Biorobotics Lab. The operator remained in the demonstration room (ie. a location remote from the robot) and controlled the robot through the Internet. Also, the operator had the option to run the simulation of the test on his/her workstation (Operator PC) at the same time as he/she was controlling the robot in the lab (Figure 3.7).

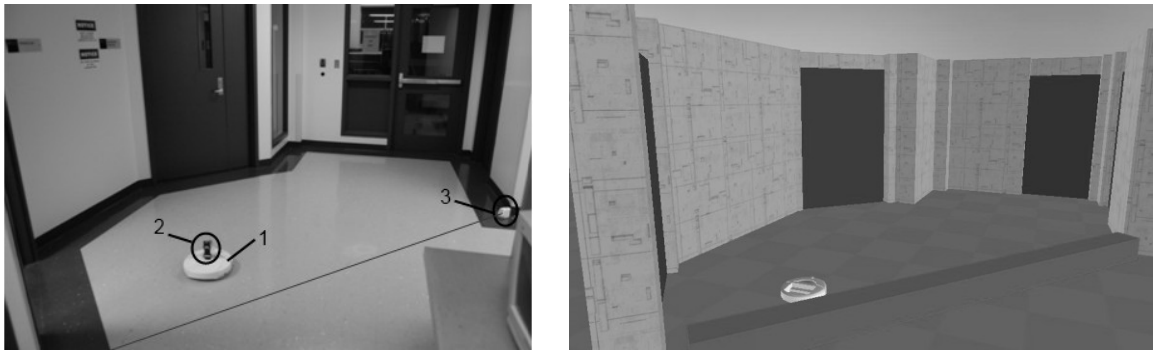


Figure 3.7. Left side image shows the area in Robotics Lab. In this picture iRobot Create (1), the wireless webcam and the Bluetooth device (2), and a virtual wall (3) can be seen. The right side image shows the simulated robot in the simulation of the scene. This simulation was running on the Operator PC.

On the Operator PC, the operator could see the video stream from the actual robot at the same time of the video stream from the simulator. So, for instance, when the robot got close to a door it appeared in both video-streaming windows. This helped the operator to have a better understanding of the robot's location when there was latency in receiving video streams from the robot. In other words, the simulation provided feedback about operator's commands, which allowed him/her to know when to turn, slow down or stop the robot (Figure 3.8).

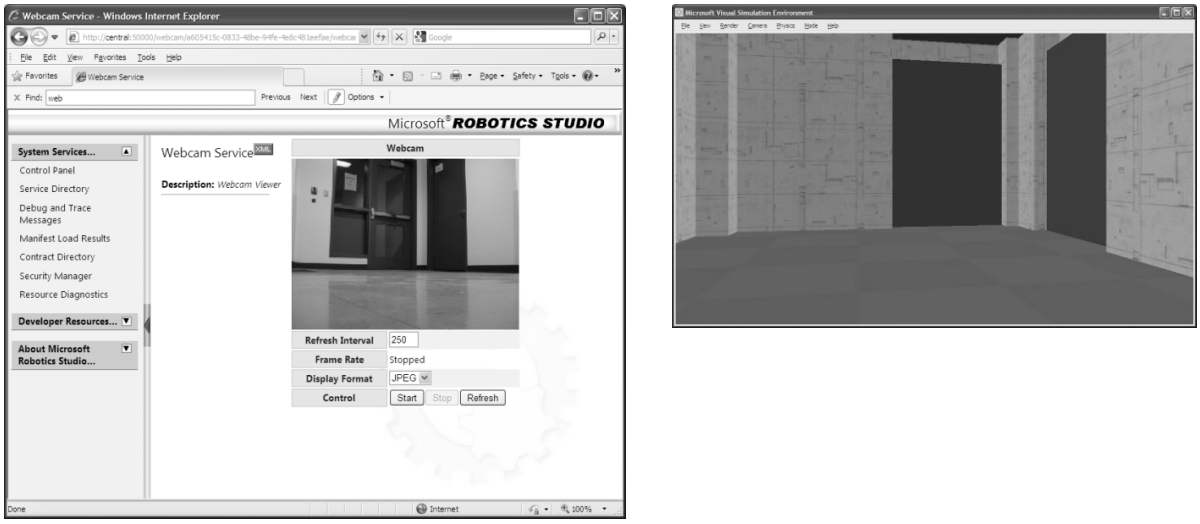


Figure 3.8 Video streaming windows on the Operator PC. On the left is the Operator PC pane which is displaying the actual image from the webcam on the iRobot Create. On the right is the simulated image.

## 3.4 CONCLUSION

This experiment revealed some challenges and issues but none of them were an obstacle to further progress. At the time of this phase there were no MRDS books published and online materials were not enough to cover all challenges and unexpected errors. The major challenge was to learn MRDS using limited available learning materials and rely more on trial and error. As a result, some issues were delaying the project for days.

The first issue happened when the VPL program was tested on the Onsite PC. There were not enough resources on the PC (CPU and Memory) to support VPL and the program

crashed. To overcome this issue the VPL program was converted to a service and directly launched. There is a built in tool in VPL that allows programs developed in VPL to be compiled as a service.

The other issue was synchronization of the actual robot and its simulated model. The speed over the ground of the actual robot could not be accurately synchronized with the speed of the simulated robot especially when the robot was operated at its full speed. So after a while the actual robot ended up in a different position than the simulated robot. A possible solution proposed was to read the position of the actual robot and update the simulated robot in a constant time interval. However this was not pursued at the time.

Overall, the experiment was successful and at this point it was concluded that MRDS was a good choice for implementation of a more advanced robotics project.

# CHAPTER 4

# PAN-TILT CAMERA SURVEILLANCE SYSTEM

In this chapter the process of developing the first part of the main project is explained. Similar to the pilot project from chapter 3, a hardware component and its simulated model were designed to be controlled simultaneously by an operator. This chapter explains how a pan-tilt camera and its simulated model can work together through MRDS. In the final project the goal was to mount the pan-tilt camera to a cable-suspended robot and stream its video to the operator.

This project required a simulated model of the camera to first be created because MRDS does not include a simulated pan-tilt camera. The model was built based on a standard pan-tilt camera, which is used in this project. A MRDS service was already available for the controller board of the actual pan-tilt camera, so to control the actual pan-tilt camera it only required development of a drive service. In addition, a drive service was developed to control the simulated pan-tilt camera.

## 4.1 THE ACTUAL PAN-TILT CAMERA

A pan-tilt camera was used in this project. The pan-tilt camera is an articulated, motor-driven arm with two Radio Controlled (R/C) servos, which give the camera two degrees of freedom. This system (Figure 4.1) includes a wireless webcam at the end of the arm. The camera has a built-in antenna that streams out videos as 2.4 GHz radio waves.

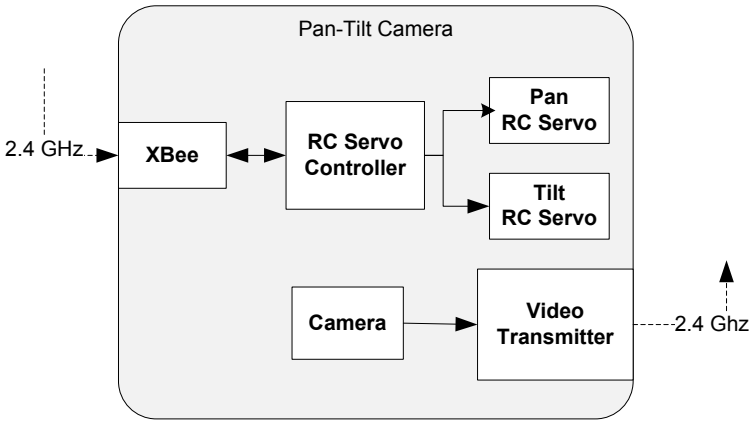


Figure 4.1. The schematic diagram of the actual pan-tilt camera

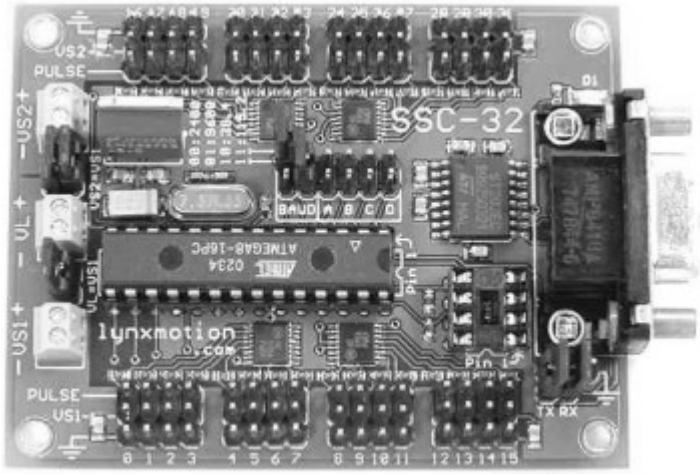


Figure 4.2. SSC-32 servo controller board (Lynxmotion, 2007)

The servos used in this camera are controlled by a SSC-32 controller board (Figure 4.2). The SSC-32 is a servo controller board manufactured by Lynxmotion that can control up to 32 servos. This controller can rotate radio-controlled (R/C) servos with positioning resolution of 0.09°/unit (180°/2000). The minimum position value 500 corresponds to 0.50ms pulse (-90°), and the maximum position value 2500 corresponds to a 2.50ms pulse (+90°). The servos are centered at a position value of 1500. A one unit change in position value produces a 1us (microsecond) change in pulse-width, which represents 0.09°. PW (Pulse Width) commands are sent to this controller via a RS232 port (Lynxmotion, 2007).

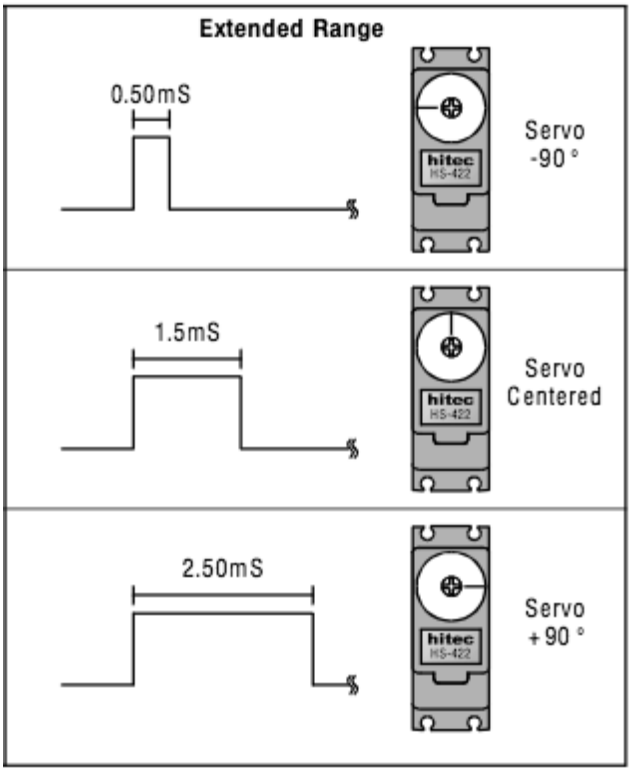


Figure 4.3. Pulse-Widths (in milliseconds) and their corresponding servo angles (Lynxmotion, 2007)

This project used a ZigBee module (XBee) to wirelessly connect the PC to the Pan-tilt camera. XBee module is a cost-effective solution for wireless connectivity to devices in a ZigBee mesh network (Digi International Inc., 2014).

### 4.2 THE SIMULATED MODEL

The simulated model of this articulated arm was created in the C# language. The main components of this model are its joints and how they are defined and attached to the other parts of the articulated arm. Both pan and tilt joints are defined as simple sphere joints with a single angular degree of freedom unlocked. As illustrated in the physics view of the simulated model (Figure 4.4), the pan-tilt camera entity consists of two joints, two segments (simple capsule shapes), a fixed base, and a sphere at the end of the second segment, which represents the body of the wireless camera.

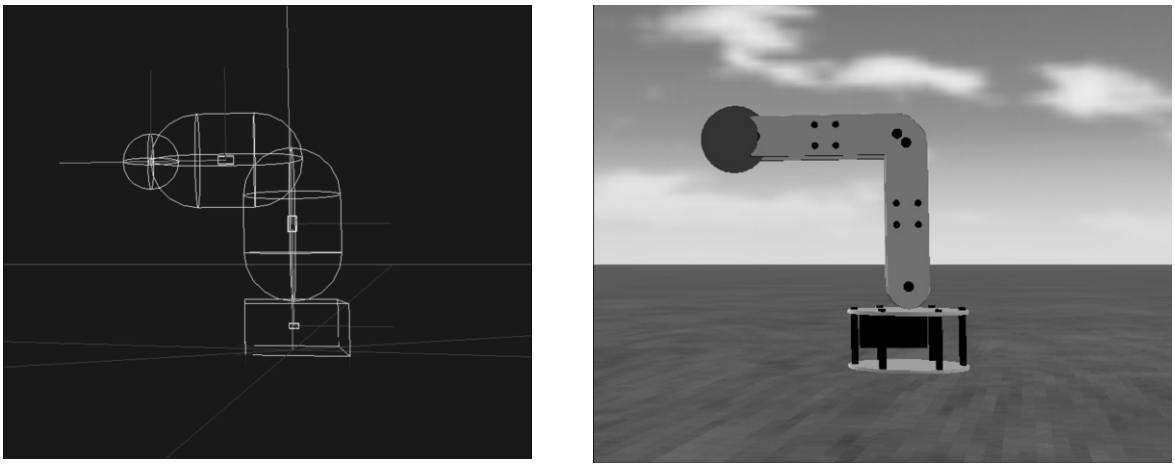

Figure 4.4. The physics view (left image) and the final simulated model after adding the mesh (right image)

In the next step a mesh was associated to every part of the pan-tilt camera to give it a more realistic look. The simulated model was created based on the primary actual pan-tilt camera. However, the actual pan-tilt camera has been redesigned, so the camera used in the final project looked different than the primary design.

As the last step in creating the simulation, a camera entity was added to the arm, and its point of view, angle of view and other properties adjusted to make its video output similar to the video output of the actual camera.

### 4.3 IMPLEMENT SERVICES

After both the simulated and actual pan-tilt cameras were setup, services for controlling them were implemented in the C# language. Services used in this application are shown in the following diagram (Figure 4.5). The core services are drive services for the actual pan-tilt camera (`PanTiltArmDrive`), and simulated pan-tilt camera (`SimPanTiltArmDrive`). The first service communicates with the actual pan-tilt camera through the `SSC32` service, which translates the requests to commands known to the controller board of the pan-tilt camera. The latter service interprets the requests to alter the orientation of simulated joints and render the simulated arm accordingly.

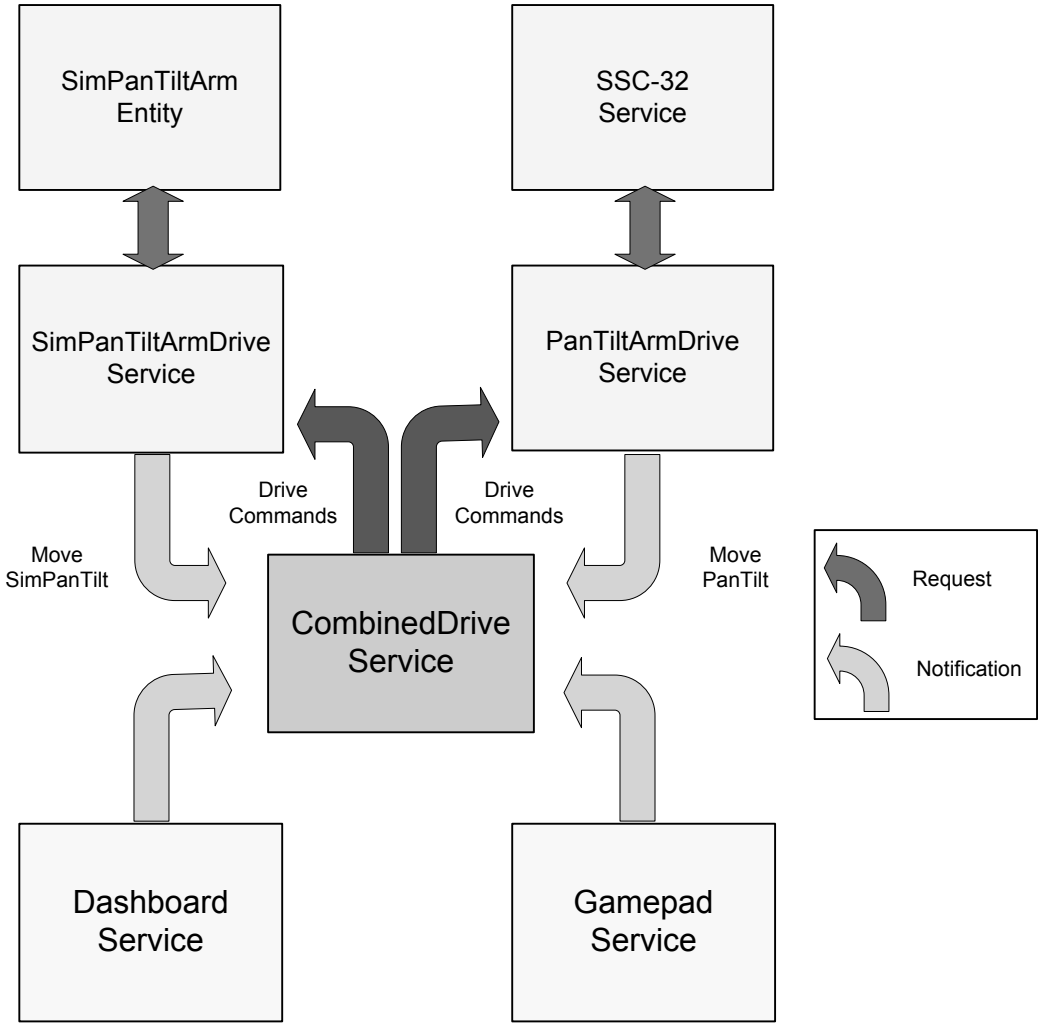


Figure 4.5. Pan-tilt camera services and their relations

Both services are implementations of the Generic Articulated Arm service. Use of this generic service allows the other services to communicate with pan-tilt drive services independent of implementation of their operation handling methods. This implementation allows the same drive commands to be forwarded to both drive services without consideration of how the handler methods are implemented in each service.

The `CombinedDrive` service is also an implementation of the Generic Articulated Arm. Upon changes in state of the `Gamepad` service or the `Dashboard` service, a notification is sent to the `CombinedDrive` service (Figure 4.5). As a result, a request is sent to both drive services to update the position of joints based on the new state of the `Gamepad` or

`Dashboard`. In return, notifications are triggered from the drive services, sending back the new state of pan and tilt joints to the `CombinedDrive` service.

### 4.3.1 Articulated Arm Drive Services

`SetJointTargetPose` is the operation message that is sent to both drive services. This message carries the orientation of each joint and in each drive service a handler method is implemented to use this information and update the joints accordingly. The rotation scale of both drive services is set in a configuration file, which is editable through the Dashboard. In the next section Dashboard and its options are explained in more details.

In `PanTiltArmDrive`, the requested angle of each joint is first converted to a servo angle then a pulse width before it is sent as a command to the SSC32 service.

Code snippet 4-1 implementation of SetJointTargetPose Handler method in PanTiltArmDrive:

```
/// <summary>
/// Set Joint Pose
/// </summary>
/// <param name="update"></param>
/// <returns></returns>
/// <remarks>Sets the pose of a single joint (in most cases this is just an angle)</remarks>
private IEnumerator<ITask> SetJointPoseHandler(armproxy.SetJointTargetPose update)
{
   // Find out which channel it is, based on the joint name
   string name = update.Body.JointName;
   int index = _channelLookup[name];

   // Figure out the desired angle in degrees
   AxisAngle orientation = update.Body.TargetOrientation;
   float orientationAngle = -orientation.Angle * Math.Sign(orientation.Axis.X);
   float jointAngle = Conversions.RadiansToDegrees(orientationAngle);
   int moveTime = 1000;

   // Get the joint
   // Throws an exception if the name does not exist
   Joint j = _jointLookup[name];

   int servoAngle = JointAngleToServoAngle(index, (int)(orientationAngle * 180 / Math.PI));
   int pulseWidth = AngleToPulseWidth(servoAngle);

   ssc32.SSC32ServoMove moveCommand = new ssc32.SSC32ServoMove();
   moveCommand.Channels = new int[1] { index };
   moveCommand.PulseWidths = new int[1] { pulseWidth };
```

```
    moveCommand.Speeds = new int[1] {(int) physicalmodel.Vector3.Length(
ProxyConversion.FromProxy(j.State.Angular.DriveTargetVelocity))};
    // Always apply some sort of reasonable time so that the arm does
    // not move suddenly
    moveCommand.Time = moveTime;

    ssc32.SendSSC32Command command = new ssc32.SendSSC32Command(moveCommand);
    _ssc32Port.Post(command);
    yield return Arbiter.Choice(command.ResponsePort,
            delegate(ssc32.SSC32ResponseType response)
            {
            update.ResponsePort.Post(DefaultUpdateResponseType.Instance);
          },
     delegate(Fault fault)
         {
          update.ResponsePort.Post(fault);
         }
      );

        yield break;
     }
```

In the `SimPanTiltArmDrive` service, a task that sets the new angle of the simulated joint is posted to `DeferredTaskQueue` after converting from radians to degrees so it can be executed when the next entity's Update method runs again. To understand the purpose of posting this task the definition of frames in the Microsoft Visual Simulation Environment must be first reviewed.

The simulator divides time into separate chunks called frames. At the start of each frame, the simulator calculates the simulator time for that frame. First, the simulator retrieves the previous frame results from the physics engine, the simulation engine used in VSE to solve object dynamics and collisions. Then the `Update` method is called for every entity in the scene. After all of the `Update` methods have completed, the scene is rendered once for each real-time camera in the scene. Finally, the scene is rendered from the eyepoint of the Main Camera, and the physics engine, begins processing the next frame (Johns & Taylor, 2008).

In the Microsoft Visual Simulation Environment, the `Update` method of an entity is called when the physics engine is not processing the frame. So this is the only time it can be guaranteed that physics engine is not busy and its methods such as `SetPose`, `SetLinearVelocity`, `SetAngularVelocity`, `ApplyForce`, and `ApplyTorque` can be

called. In `SimPanTiltArmDrive` service since the angles of the joints have to be updated outside of `Update` method, first a task was created to take care of this change and this task was added to `DeferredTaskQueue`. Tasks of this queue are executed when the `Update` method is called. With this technique tasks that are dependent on the physics engine's methods can be called anywhere in the program but execution of them will be deferred to the time when the physics engine is not busy.

Code snippet 4-2 Implementation of SetJointTargetPose Handler method in SimPanTiltArmDrive:

```
/// <summary>
/// Set Joint Pose
/// </summary>
/// <param name="jointPose"></param>
/// <returns></returns>
/// <remarks>Sets the pose of a single joint</remarks>
[ServiceHandler(PortFieldName = "_articulatedArmPort")]
public void SetJointTargetPoseHandler(articulatedarm.SetJointTargetPose jointPose)
{
   DOFDesc dof = _state.JointDir[jointPose.Body.JointName];
   JointDesc desc = dof.Description;
   switch (dof.Type)
      {
        case DOFType.Twist: desc.TwistAngle = (float)(jointPose.Body.TargetOrientation.Angle * 180 /
Math.PI); break;
        case DOFType.Swing1: desc.Swing1Angle = (float)(jointPose.Body.TargetOrientation.Angle *
180 / Math.PI); break;
        case DOFType.Swing2: desc.Swing2Angle = (float)(jointPose.Body.TargetOrientation.Angle *
180 / Math.PI); break;
         }

   PhysicsJoint thisJoint = (PhysicsJoint)desc.JointEntity.ParentJoint;

   // A task with the joint information is created then posted to
   // DefferdTaskQueue, letting the entity to update the joint, by
   // calling SetDriveInternal method, once the physics engine is stopped.
   Task<PhysicsJoint, Quaternion, Vector3> deferredTask =
            new Task<PhysicsJoint, Quaternion, Vector3>(thisJoint, desc.JointOrientation,
desc.JointPosition, SetDriveInternal);
        _entity.DeferredTaskQueue.Post(deferredTask);
}

/// <summary>
/// Set Drive Internal
/// </summary>
```

```
/// <remarks>Sets the position and orientation of a joint</remarks>
void SetDriveInternal(PhysicsJoint joint, Quaternion orientation, Vector3 position)
{
   if (joint.State.Angular != null)
          joint.SetAngularDriveOrientation(orientation);
   if (joint.State.Linear != null)
          joint.SetLinearDrivePosition(position);
}
```

As mentioned earlier, the `CombinedDrive` service receives operation messages and forwards them to both pan-tilt drive services. The following code shows how this was implemented.

Code snippet 4-3 Implementation of SetJointTargetPose Handler method in CombinedDrive:

```
[ServiceHandler(PortFieldName = "_articulatedArmPort")]
public virtual IEnumerator<ITask> SetJointTargetPoseHandler(articulatedarm.SetJointTargetPose
onApply)
{

  for (int i = 0; i < _drivePorts.Count; i++)
    {
       _drivePorts[i].Post(onApply);
    }

  yield break;
}
```

In this code, `_drivePorts` is a list of the `ArticulatedArmOperations` containing ports for both Pan-tilt drive services.

On another matter, services working with Microsoft Visual Simulation Environment are dependent on the existence of entities in the simulated scene. In this case, the simulated pan-tilt drive service is dependent on the existence of the simulated joints to initialize its state by reading the properties of the simulated joints such as their names and types.

In writing the simulated pan-tilt drive this point is considered and unlike regular services this drive service is not started right after it is called. The service waits until the simulated engine notifies it that the pan-tilt simulated entity is inserted into the scene then it starts. If this is not considered in implementation the drive service does not load successfully.

### 4.3.2 Dashboard Service

The `Dashboard` service is a modified version of the `Simple Dashboard` service, supplied with MRDS. The service has a user interface that helps the operator to communicate with drive services of robots and modify some of the control settings. This service can communicate with any implementation of Generic Articulated Arm service. It can connect to a host where MRDS is running on, detect any generic drive services and list them for the operator. The operator can then select a drive service and send control commands to it using the trackball or arrow keys on the Dashboard window, or via a Gamepad. The `Gamepad` service is launched by the `Dashboard` service as soon as the service is started.

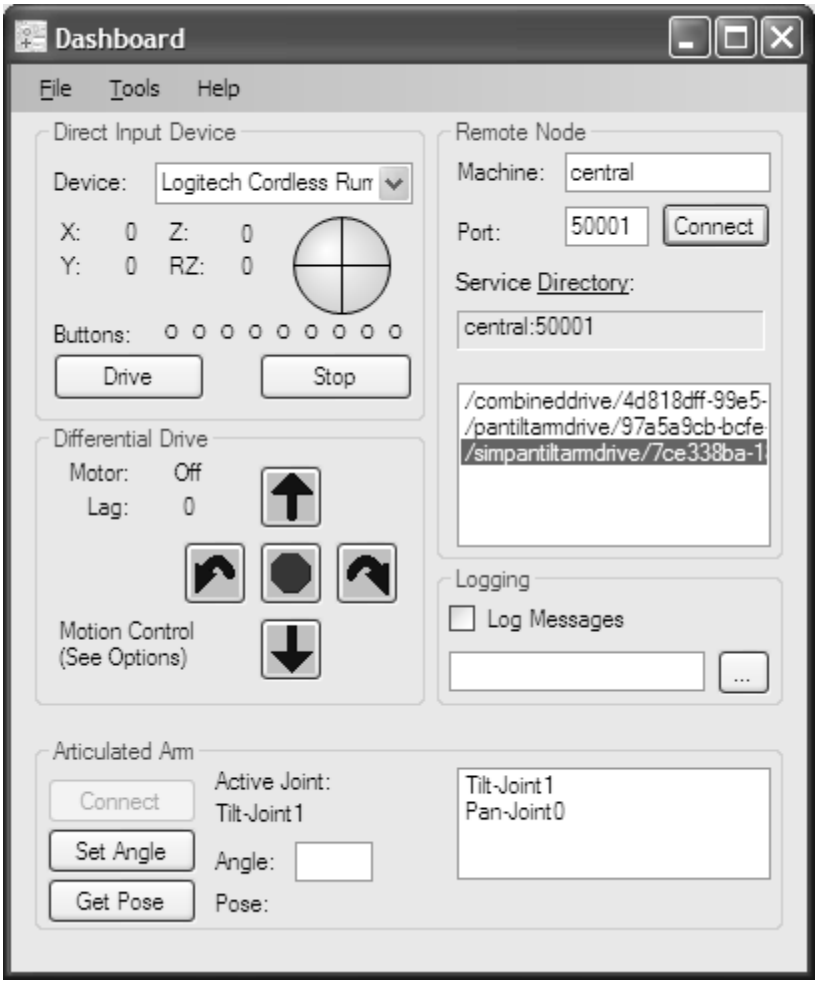


Figure 4.6. Dashboard user interface lets the operator control the cameras and move the pan and tilt joints separately by setting their angles manually, as noted in the Articulated Arm section.

Under the Tools menu the operator can select Options to access the settings. As illustrated in Figure 4.7, under `Pan-tilt Arm`, the operator can set the max pan and tilt angles of pan and tilt joints as well as the rotation scale (in degrees) and rotation speed (in milliseconds).

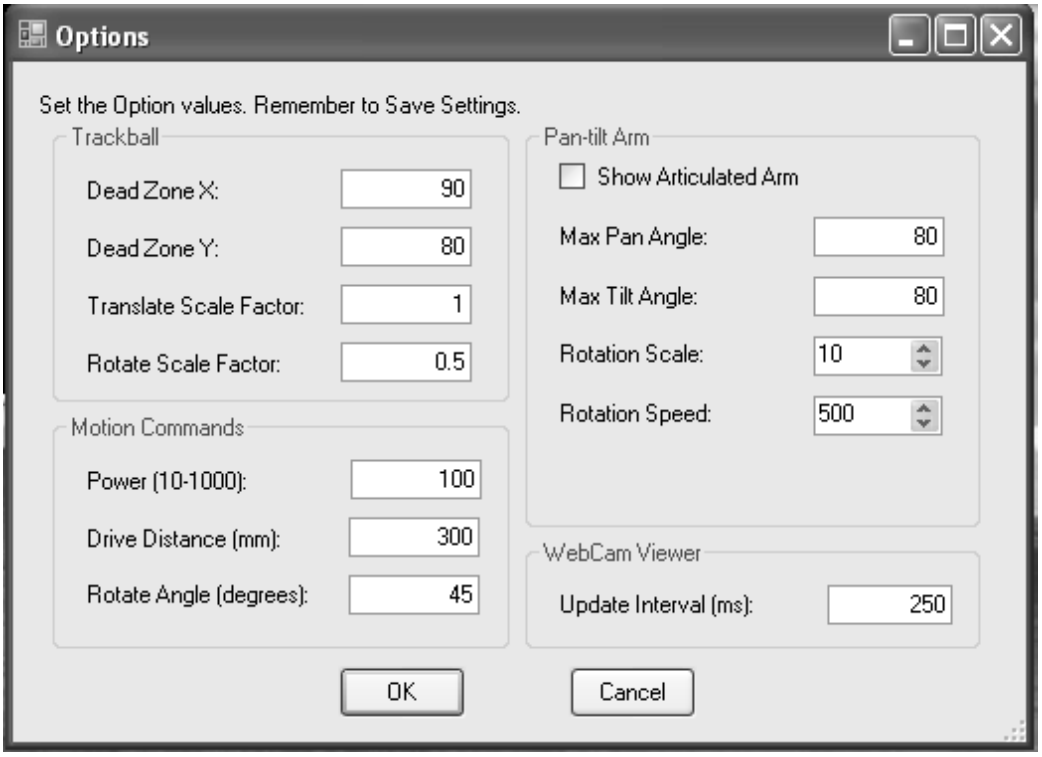


Figure 4.7. Settings can be modified in the Options window, accessible from the Dashboard menu.

The right analog stick is the stick dedicated to control of the Pan-Tilt Camera in this project. As it is explained in the next chapter, the left analog stick, which generates values of X and Y axes, is selected to control the wire-suspended robot.

The `Max Pan Angle` and `Max Tilt Angle` are used to set thresholds on rotation angles of Pan and Tilt joints. In the `Dashboard` service these thresholds are checked when the rotation angles of joints are incremented or decremented. Once a threshold is reached, the service ceases incrementing or decrementing that angle.

The other options used in this project are dead zones, under `Trackball`. The dead zone is a region where the movement of the gamepad's analog stick has no effect. The values of both X and Y axes can be an integer number between -1000 to +1000. By setting the dead zones to a number greater than 0 any received value lower than the dead zone values are going to be ignored. This is especially helpful for use of gamepads that do not return to zero when their sticks are released. In the above example (Figure 4.7), `DeadZone X` is set to 90. That means any received input from the gamepad on X axis is going to be considered as 0 unless it is greater than 90 or less than -90.

Once the OK button is pressed, the new settings take effect in both the simulated and actual pan-tilt cameras on the fly.

### 4.4 PROGRAM EXECUTION AND OPERATION

The following schematic diagram shows how the system was set up (Figure 4.8). The operator was at the Operator PC and could see the camera view from both the actual camera at the site and the camera in the Visual Simulation Environment at the same time. The Dashboard running on the Operator's PC allowed the operator to control both cameras simultaneously. Also the operator has the option to use a wireless Gamepad to move the pan-tilt cameras.

Control commands were sent to the drive service that is running on the Onsite PC (`PanTiltArmDrive`) and from there they are forwarded to the Pan-tilt camera through an XBee module. The Onsite PC also received video streams from the scene, through the Webcam service, and transferred them to the Operator PC. The distribution of service was demonstrated by running `PanTiltArmDrive` and `Webcam` services on the Onsite PC.

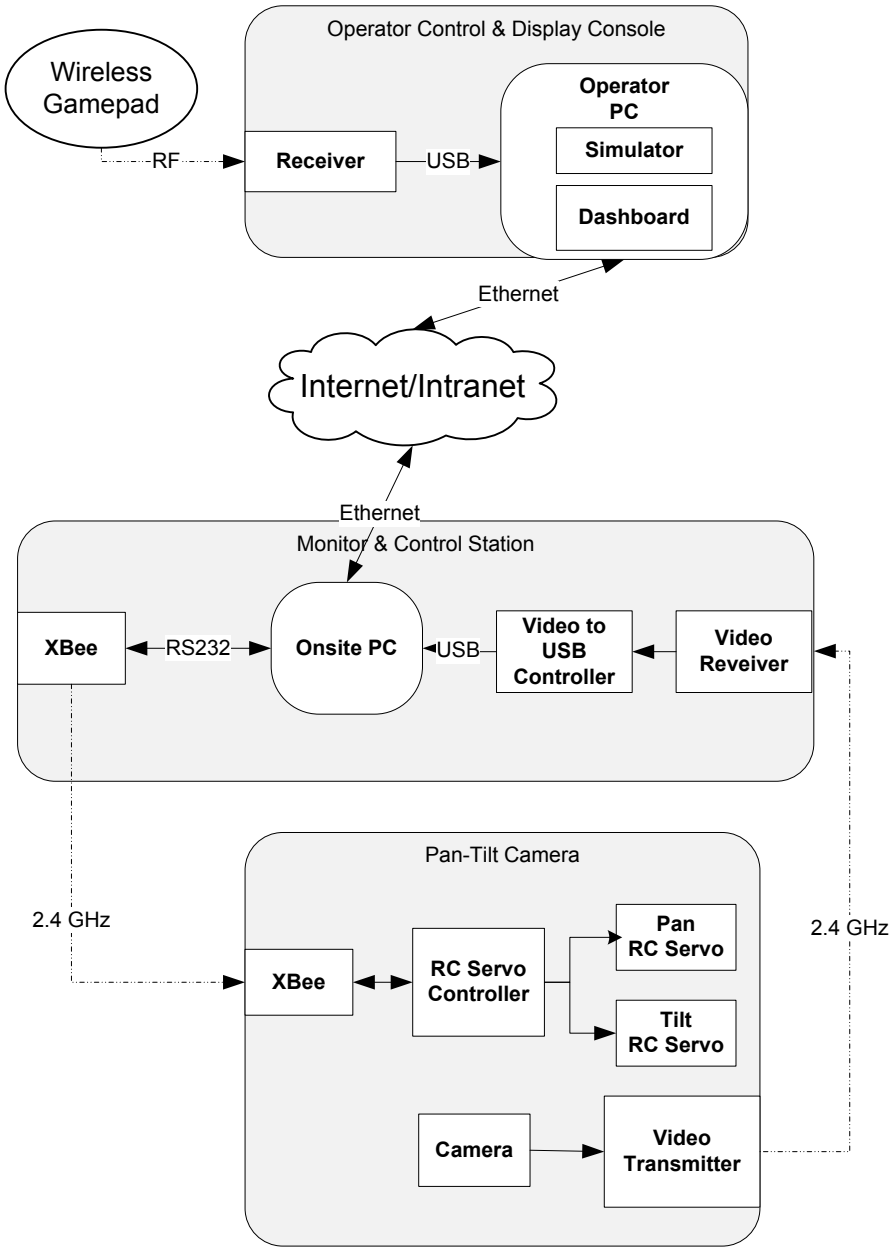


Figure 4.8. Schematic of the system for controlling the pan-tilt camera

The following images (Figure 4.9) illustrate what the operator could see on the Operator PC. The operator has two camera windows open next to the Dashboard User Interface. The operator also has an option to select either the actual or the simulated drive service, one at a time, to control one of them only without moving the other. This option can help for testing and adjusting purposes.

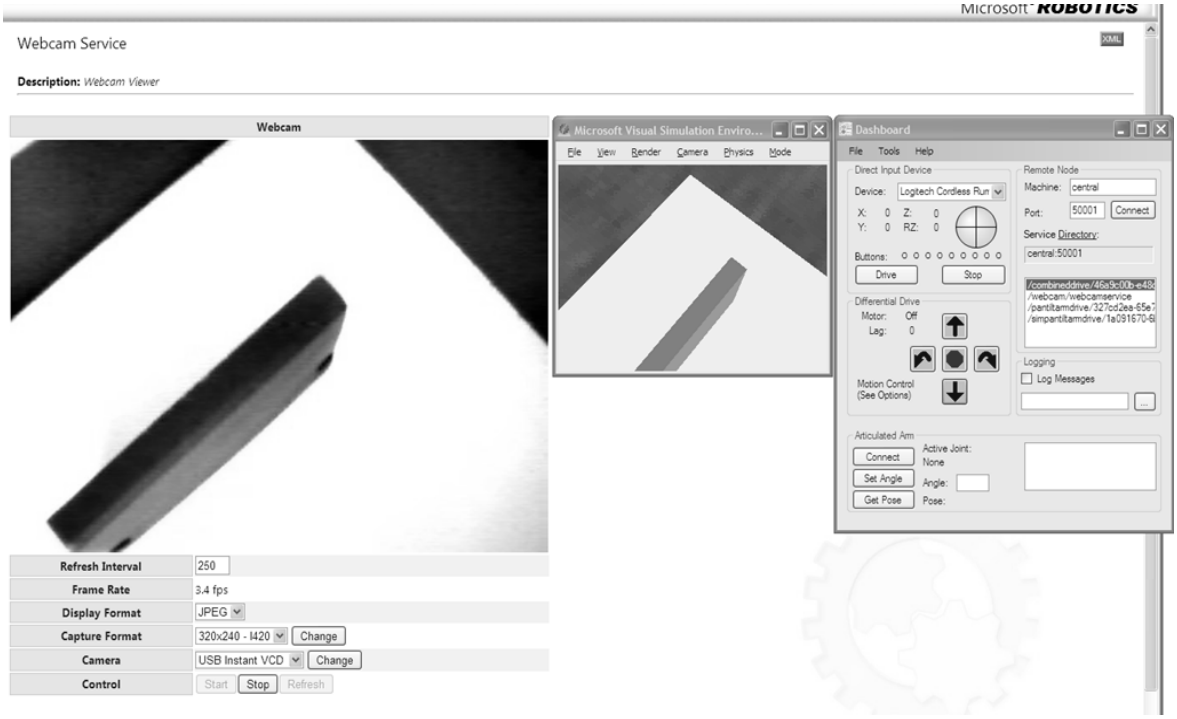


Figure 4.9. A screenshot of Operator PC shows the Dashboard, Simulated Webcam, and Actual Webcam windows. A top view of the notepad and part of the mounting stand is captured by both webcams.

By turning left, right, up and down the operator can look around. The following images (Figure 4.10) illustrate the actual and simulated scenes.

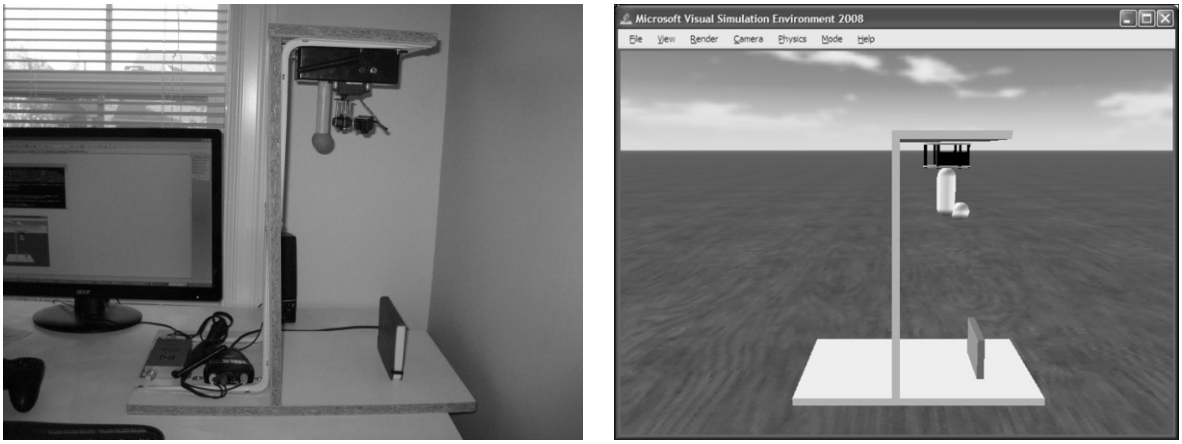


Figure 4.10. Pan-tilt camera mounted to the mounting stand

## 4.5 Challenges and Future Work

One of the challenges was the hard coded settings defined inside of the SSC-32 service. This service did not have an option to save its configuration settings in an external file, requiring recompiling the code every time the settings had to be changed. To avoid the requirement of recompilation, the code was modified to support this option. The standard configuration method of .NET framework does not work on DSS so the state of service was saved in an XML file. The state also was retrieved from the same XML file and replaced with the current state of the service anywhere in the code it was required.

Code snippet 4-4 Implementation of a solution to save state of the service in a configuration file:

```
// Enabling use of Configuration file
System.Uri InitialStateUri(string initialStateString)
{
  System.Uri storeAddr = new System.Uri(new Uri("http://localhost/"), ServicePaths.MountPoint +
"/");
  return new System.Uri(storeAddr, initialStateString);
}

// Saving state to a config file
void SaveStateToFile(string filename)
{
  //first find the partner that holds your state service
  PartnerType pt = FindPartner(
            Microsoft.Dss.ServiceModel.Dssp.Partners.StateService,
            ServiceInfo.PartnerList);
```

```
  //create mount point uri:
  System.Uri thisUri = InitialStateUri(filename);

  //using the URI above, issue a Replace to the mount service.
  // You can also modify this uri to create various back-ups when you think it is appropriate.
  // So instead of SaveState(), just issue the Replace directly to the mount service using the URI/file.
  mount.MountServiceOperations port =
ServiceForwarder<mount.MountServiceOperations>(thisUri.AbsoluteUri);

  DsspDefaultReplace replace = new DsspDefaultReplace(_state);
        port.Post(replace);
}
```

There is no standard solution for synchronized driving in MRDS. In this project a third drive service is used as a repeater of drive commands to the simulated and actual drive services, allowing them to move simultaneously. The drive service is called the `CombinedDrive` and, as it was explained in the previous part, it is an implementation of Generic Articulated Arm only to forward controlling commands. `ServiceForwarder` was used in the implementation of this service to establish ports for communication with pan-tilt drive services.

Code snippet 4-5 During initialization, ConnectDrive is called for every Articulated Arm Drive service

```
foreach (ServiceInfoType info in list)
{
   // If this service is an Articulated Arm Drive and it is not this service then add it to the list of ports
   if (info.Contract == articulatedarm.Contract.Identifier &&
!info.Service.Contains("combineddrive"))
     {
        // Connect to the drive and add it to the list of ports
        yield return Arbiter.ExecuteToCompletion(
          Environment.TaskQueue,
           new IterativeTask<string>(info.Service, ConnectDrive)
         );
     }
 }
```

Code snippet 4-6 ConnectDrive adds the port of the Articulated Arm Drive service to the list of all found services` ports, _drivePorts.

```
/// <summary>
/// Connect to a Generic Articulated Arm Drive service
```

```
/// </summary>
/// <param name="service"></param>
/// <returns>Updates the global _drivePorts list</returns>
IEnumerator<ITask> ConnectDrive(string service)
{
  articulatedarm.ArticulatedArmOperations drivePort = null;

  // Get a new articulated arm drive port
  drivePort = ServiceForwarder<articulatedarm.ArticulatedArmOperations>(service);

  if (drivePort != null)
  {
    // Record the service URI so the user can see it
   LogInfo(LogGroups.Console, "Connected to Drive " + service);

    // Remember the port
    _drivePorts.Add(drivePort);

  }
  yield break;
}
```

Debugging of services codes were also challenging. As Martin Fowler warned in his book, loosely coupled services can easily become "An architect's dream and a developer's nightmare" (Fowler, 2002). Due to the use of loosely coupled services in programming under MRDS and involvement of several services in this project to accomplish a task, a bug in one service could affect the final result. To find the bug it was required to trace the issue back between several services until it could be found. An example is the issue that happened in one of the functionalities of the `Dashboard` service. Upon selecting an articulated arm drive in the Dashboard, all joints of the arm are supposed to be listed under Active Joints text box (Figure 4.6). It stayed blank when the first `PanTiltArmDrive` was selected. After several hours of debugging it turned out that there was a problem in the Pan-tilt camera circuit board and the query command sent to the SSC-32 controller was returned Null. The Dashboard service sends a Get message to the articulated drive, requesting its state. This request then is sent to SSC32 service and then to the SSC-32 controller. In this example, the error happened at the last service.

One of the things that should be implemented in future for a more accurate result is to update the SSC32 config file every time that the SSC32 service's state is changed, for each servo move. Therefore the simulator can read the file and sync up if it is not synchronized.

### 4.6 Conclusion and Summary

By the end of this part of the project, development of full customized MRDS services was done successfully. The pan-tilt drive services and the simulated camera entity were developed in C# from scratch. Development of services for MRDS was challenging and it required learning the MRDS fundamentals in detail. No out of box solutions were available for some of the functionalities required in this phase and it was required to reach out to MRDS forums for help.

Synchronized move of the simulated pan-tilt camera and the actual pan-tilt camera was the other feature of MRDS which was tested successfully in this phase. This could happen by sending the same control commands and adjusting the speed of movement of the simulated joints and the RC servos.

In the end, another important feature of MRDS, distribution of services, was also tested by running the `PanTiltArmDrive` and Webcam services on the Onsite PC while other services were loaded on the Operator PC.

# CHAPTER 5

# WIRE-SUSPENDED VIDEO SURVEILLANCE SYSTEM

After the successful development and testing of the pan-tilt camera video surveillance system, the wire-suspended video surveillance system (WireCam) was implemented in the final project. The WireCam consists of four DC servo motors, four motor driven winches, four wires and a flying base, which the pan-tilt camera, from the previous chapter, is mounted to.

The system, illustrated in Figure 5.1, can be set up in a room with the four winches mounted in the four corners. Each servo drives an attached winch to wind or unwind a string/wire wrapped around the threaded drum. All four cables are hooked to a platform (the Flying-Base) which moves the mounted pan-tilt camera in a three dimensional space.

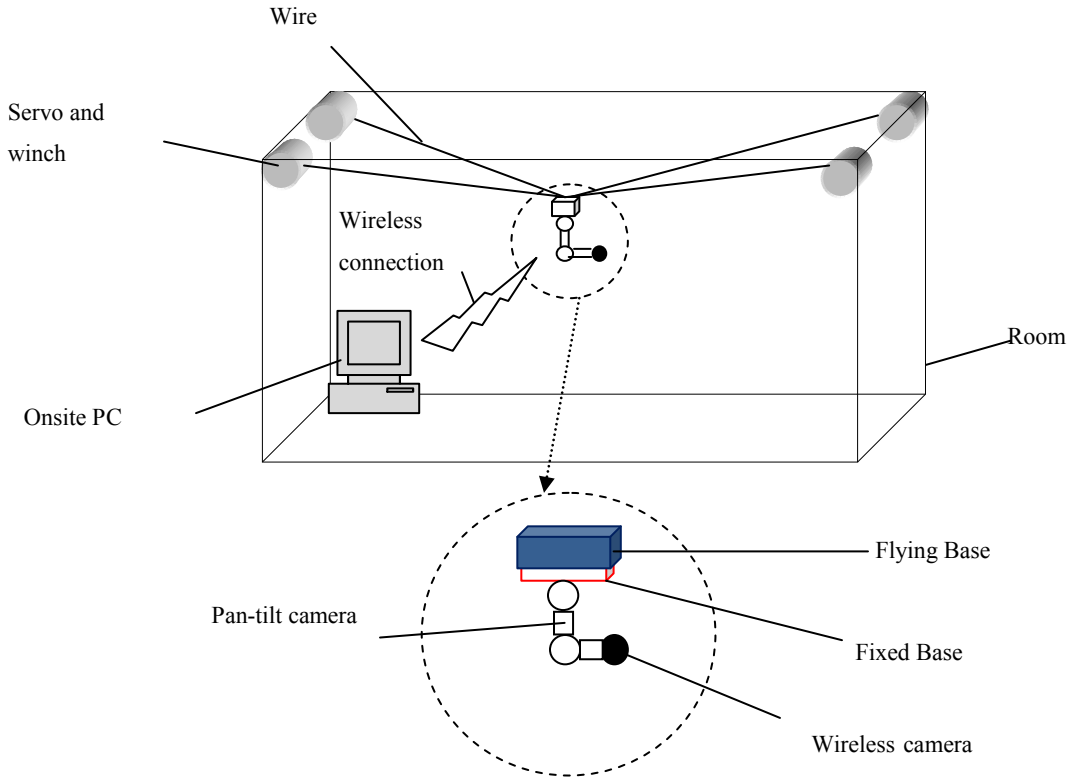


Figure 5.1. Onsite setup of the wire-suspended video surveillance system

Unlike the control board used in the pan-tilt camera (the SSC-32), there was no MRDS service available for the controller boards used in this system to drive the winches. Therefore, in addition to creating MRDS drive services for both the simulated and actual WireCam, a service was created to communicate with the controller board.

Similar to the implementation method used in the past two chapters, before getting to the actual hardware implementation the simulated model of the system was first implemented. However, since this system required a kinematic algorithm the algorithm was first tested in a C# test program as the initial step before any implementation in MRDS.

## 5.1 KINEMATICS OF THE SYSTEM

Before explaining the algorithm it is best to review some of the terms used in this section. In general, servo motors (aka servos) are closed system DC motors with built in encoders and control circuitries, which allow monitoring the angular position of each motor's shaft at anytime. Counts from the shaft encoder counter make it possible to measure the angle of rotation of the winch drum and hence length of wire out (McComb, 2011) . The constant number of `Count_per_meter` represents the counts of the shaft encoder per one meter of traveling distance. In this project, the Flying-Base refers to a platform that the pan-tilt camera is mounted on and is movable by wires connecting it to the servo driven winches in each corner of the room labeled as A, B, C and D.

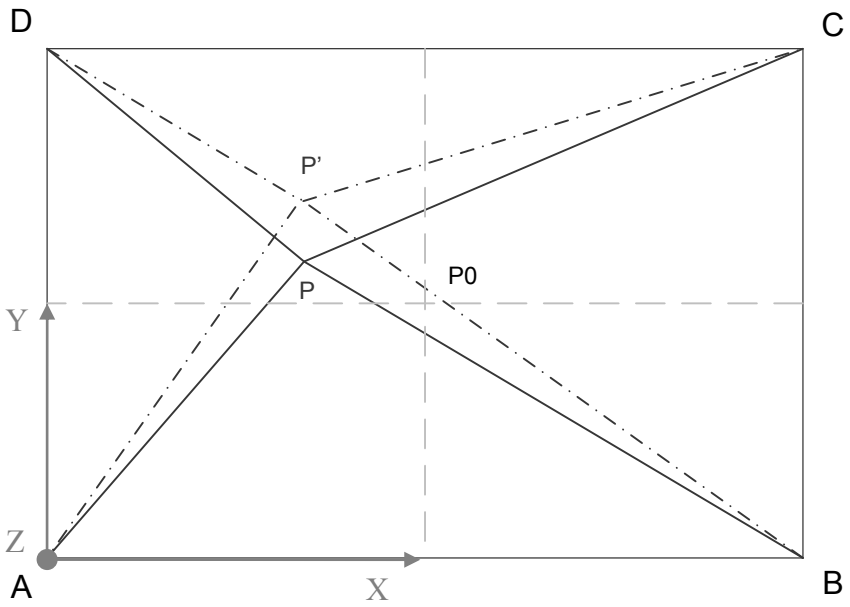


Figure 5.2. Setup of servos in a room and their labels illustrated in this figure. P represents the current location of the Flying-Base and P' is the next position calculated based on the latest inputs.

In a nutshell, this algorithm calculates the next position of the Flying-Base based on its current position and the inputs received from an input device like gamepad or a software application. The assumption is that the time for each travel is constant (T_interval) so inputs with greater values increase the velocity of the Flying-Base.

As illustrated in the following flowchart (Figure 5.3), initially the position of the Flying-Base is set to the home position, P0, which is in the middle of the room horizontally and in the highest position vertically. In the next step the distances of the Flying-Base to each servo at four corners of the room (Ap, Bp, Cp, and Dp) are calculated.

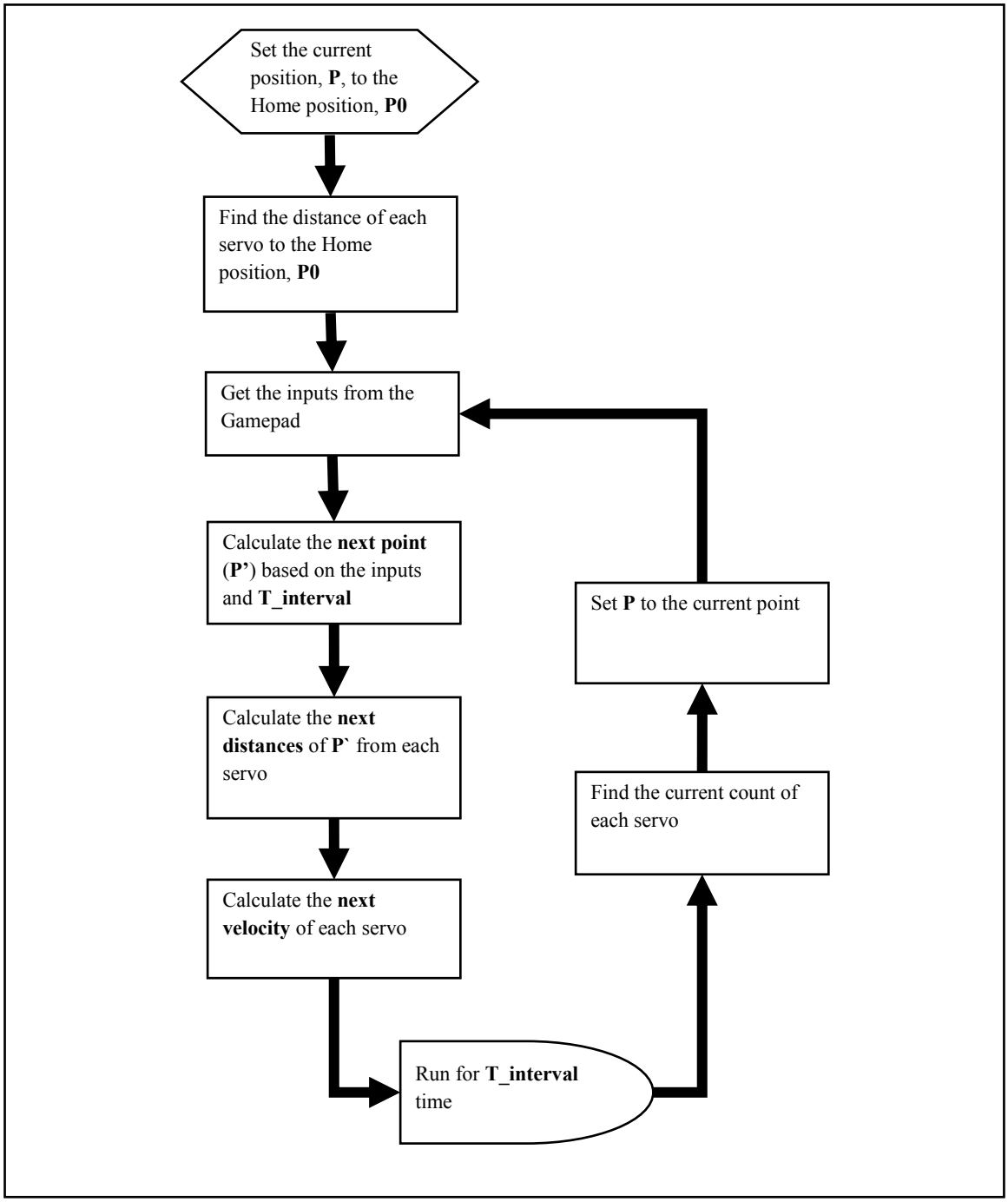


Figure 5.3. Flowchart of the algorithm used to drive the Flying-Base in 3 dimensions

Now based on the inputs received from the controller device or the application the next position of the Flying-Base, P'(Xnext, Ynext, Znext), after T_inteval seconds is calculated.

In the following equations Xinput, Yinput, and Zinput are the values of axes received from the input device. X_INPUT_MAX, Y_INPUT_MAX, and Z_INPUT_MAX are the maximum values that each axis can take. X_MAX, Y_MAX, and Z_MAX are the maximum distances the Flying-Base can travel in each direction in T_interval seconds when the input axis of that direction is set to the maximum. Xnext, Ynext, and Znext represent the coordinates of the next position of the Flying-Base after T_interval seconds. Finally, the T_inteval is the constant time allowed for the Flying-Base to move with the calculated/assigned velocity from the current position (P) to the next position (P').

$$x_{next} = \mathrm{x_{now}} + \Delta\mathrm{x} = \mathrm{x_{now}} + \left(\frac{\mathrm{Xinput}}{\mathrm{X\_INPUT\_MAX}} \times \mathrm{X\_MAX}\right)$$

$$y_{next} = \mathrm{y_{now}} + \Delta\mathrm{y} = \mathrm{y_{now}} + \left(\frac{\mathrm{Yinput}}{\mathrm{Y\_INPUT\_MAX}} \times \mathrm{Y\_MAX}\right)$$

$$z_{next} = \mathrm{z_{now}} + \Delta\mathrm{z} = \mathrm{z_{now}} + \left(\mathrm{z_{input}} \times \mathrm{Z\_MAX}\right)$$

At this point the current and next distances of the Flying-Base to each of the servos in the four corners of the room can be found. Distance of the Flying-Base to each servo is calculated by a linear algebra method implemented by using a computerized displacement measuring system modified from a paper published by Das and Kozey (1994). In this method by providing the coordinates of the Flying-Base and based on the measurements of the room, distances of the Flying-Base to each servo is determined.

$$Distance_{AP} = \sqrt{\left(\mathrm{x_p} - \mathrm{x_A}\right)^2 + \left(y_p - y_A\right)^2 + \left(z_A - z_p\right)^2}$$

$$Distance_{BP} = \sqrt{\left(\mathrm{x_p} - \mathrm{x_B}\right)^2 + \left(y_p - y_B\right)^2 + \left(z_B - z_p\right)^2}$$

$$Distance_{CP} = \sqrt{\left(\mathrm{x_p} - \mathrm{x_C}\right)^2 + \left(y_p - y_C\right)^2 + \left(z_C - z_p\right)^2}$$

$$Distance_{DP} = \sqrt{\left(\mathrm{x_p} - \mathrm{x_D}\right)^2 + \left(y_p - y_D\right)^2 + \left(z_D - z_p\right)^2}$$

Using these distances the velocity of each servo can be now calculated:

$$Velocity_A = \frac{(\text{Distance\_AP}' - \text{Distance\_AP})}{\text{T\_interval}} \times \text{Count\_per\_meter}$$

$$Velocity_B = \frac{(\text{Distance\_BP}' - \text{Distance\_BP})}{\text{T\_interval}} \times \text{Count\_per\_meter}$$

$$Velocity_C = \frac{(\text{Distance\_CP}' - \text{Distance\_CP})}{\text{T\_interval}} \times \text{Count\_per\_meter}$$

$$Velocity_D = \frac{(\text{Distance\_DP}' - \text{Distance\_DP})}{\text{T\_interval}} \times \text{Count\_per\_meter}$$

On the other hand, another method, again based on Das and Kozey (1994), is used to find the current coordinates of the Flying-Base in three-dimensional position by knowing its distances from three points with known coordinates, A, B and C. As the location of the Flying-Base depends on the length of wires connecting it to servos, servos' angular positions must be changed to change the position of the Flying-Base. The encoder count of each servo can be requested to be used to find the length of wires.

### 5.1.1 Testing the Algorithm

Before starting to implement the algorithm in MRDS services, a test application was developed in C# language to check the accuracy of the algorithm without getting other factors interfering.

The test application consists of a controller where inputs similar to what is received from the real gamepad are generated and the position of the Flying-Base in two coordinate planes is plotted, one representing X-Y coordinates and another one X-Z coordinates. Once the Gamepad Simulator is started, real-time data is updated in the text boxes to make it possible to examine them at any time to compare them with input data at the time of sampling, and to evaluate the algorithm (Figure 5.4).

The first graph is the top view of a room showing attached servos to each corner with A, B, C and D labels. Having that assumption in mind, the second graph would be the front view of the room. The red asterisks in these graphs represent the current or past location of the Flying-Base so as it moves its path would be traceable.

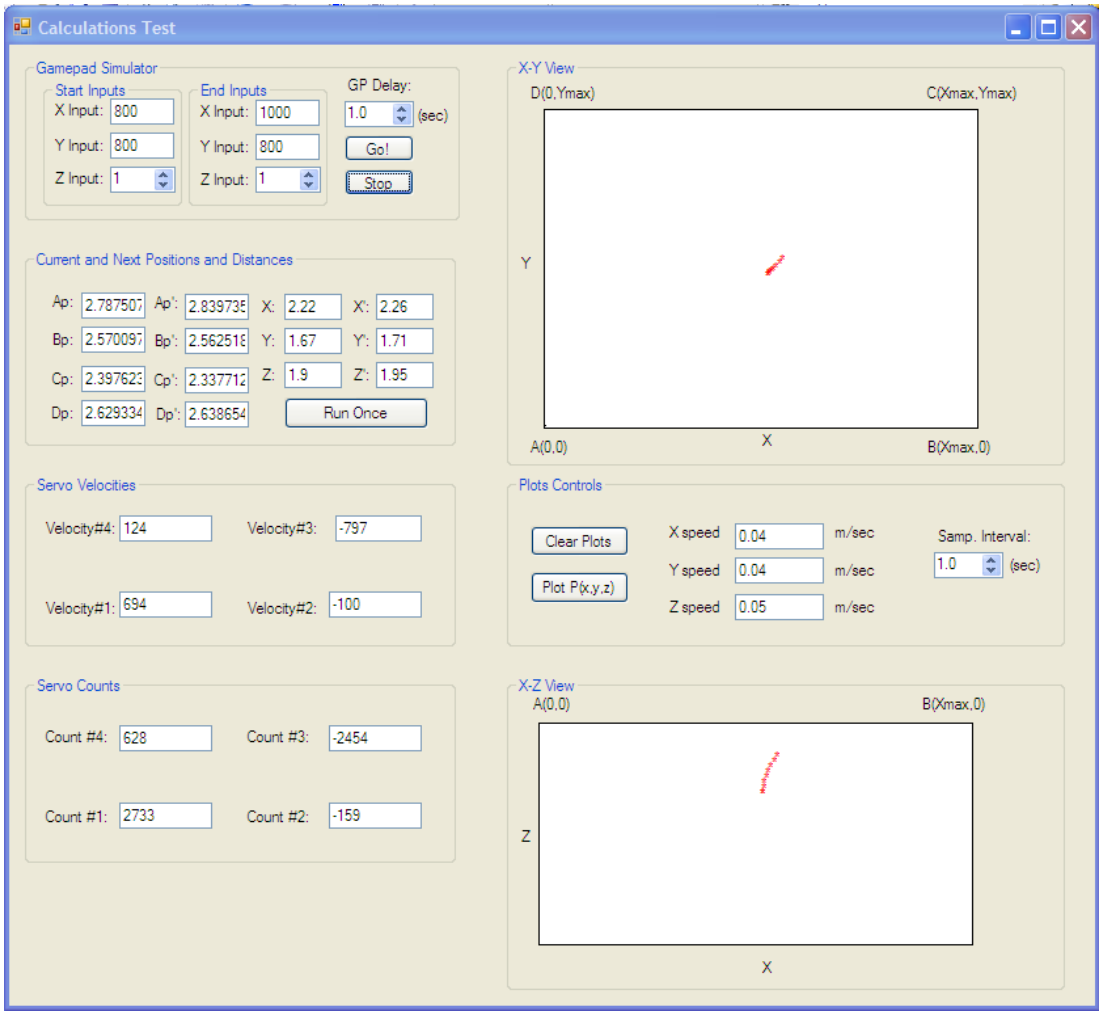


Figure 5.4. The test application allows users to try out the algorithm. Sampling Interval represents the time between each position recalculation.

The `Gamepad Simulator` simulates inputs generated by the real Gamepad for X and Y axes ranging from -1000 to 1000. As the plan for the main project is to control Z-axis direction by two gamepad buttons therefore in this application Z can be set to -1, 0, or 1 to decrease, stop, or increase Z-axis respectively.

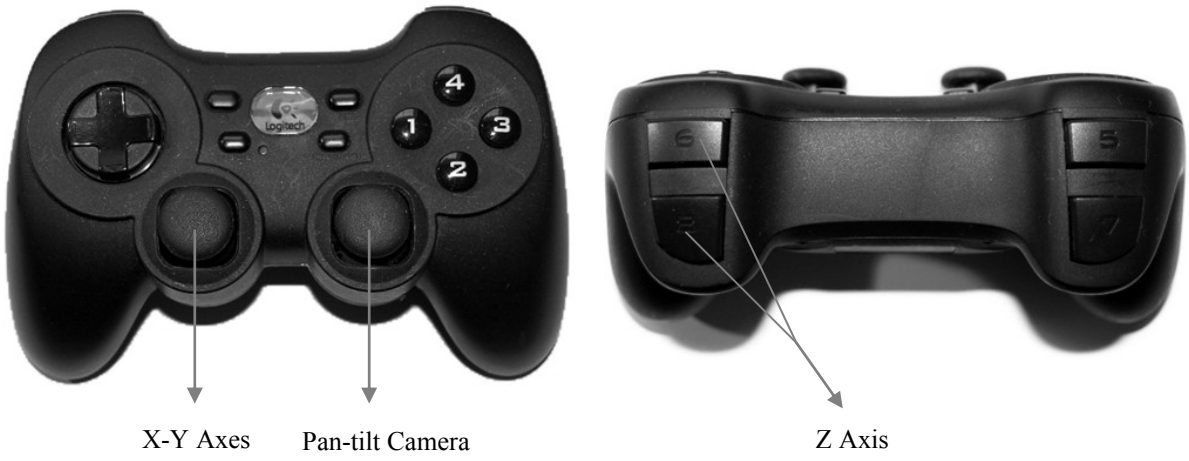


Figure 5.5. The model of the gamepad used in this thesis, Logitech Rumblepad 2, has two analog sticks. One of them is used to move the WireCam in X-Y direction and another one is for controlling the pan-tilt camera. The wire-suspended system is moved in Z direction using the right shoulder button pair.

In the `Gamepad Simulator` section there are two textboxes for each axis that let the examiner test different scenarios that could be created by a real gamepad. The user enters a starting value in the `Start Inputs` textboxes and the maximum values in `End Inputs` textboxes. For instance, to simulate sliding of analog stick on the gamepad from home position to the end corner on the X axis positive direction then the first textbox of X Input has to be set to 0 and the second one set to 1000.

Now once the user clicks on `Go!` button value of `X Input` starts increasing from 0 to 1000 gradually until the user halts the process by clicking on `Stop` button. X and Y axes in the gamepad can have values between -1000 to 1000. This is set differently for Z Input (values between -1 and 1) to match it with the implementation of the application which instead of the analog stick shoulder buttons on front side of the gamepad are going to control the robot's moves in Z direction.

The next section on this panel, `Current and Next Positions and Distances`, shows the current (Ap, Bp, Cp, Dp) and next (Ap', Bp', Cp', Dp') distances of the Flying-Base from each servo plus its current and next position of itself (p(X, Y, Z) and p'(X', Y', Z') points). By default p(X, Y, Z) is set to the middle of the room at the highest position

vertically (Zmax) however the examiner can modify the home position by setting p(X,Y,Z) to different values. Two next textbox groups continuously represent the current velocity and encoder count of each servo. Servo #1 is located at corner A and #2 at corner B and so on.

The last section, `Plots Controls`, has two Buttons, one for clearing the plots so the user can rerun a test and other to spot values of p(X,Y,Z) in plots. The Flying-Base speed is updated in textboxes in this group pad and `Samp. Interval` is located here to control the sampling time interval of positioning calculations.

Several test scenarios were performed on this application including, but not limited to, running the system with single axis input, with combination of axes inputs, and with positive and negative inputs. `Run Once` option also made it possible to examine all numbers by setting the inputs and current coordinates of the Flying-Base and calculate the next coordiates. Finally, the traceable path of the Flying-Base on the plots helped to try some tests and visually examine the result of algorithm. The tests proved that the kinematics and the algorithm were working as expected.

### 5.2 DESIGNING THE SIMULATION

Once the algorithm was tested successfully it was time to use it in action with the simulation of the system. The simulation of the WireCam was first created as a separate entity then it was added to a scene. The scene was the simulation of a room where the system was going to be installed.

A room in the iDLab, at Dalhousie University, was selected for this purpose. The first step was to design a scene, which consists of the main objects in the room so the video streams from the actual camera could be matched with what was returned from the simulated webcam. The main objects of this room were three desks, a door, a window, and a whiteboard, which were all added to the simulated scene in roughly the same sizes (Figure 5.6 and Figure 5.7).


Figure 5.6. View of the simulated room from one of the corners. The pan-tilt camera can be seen in the middle of the room, mounted to the Flying-Base.

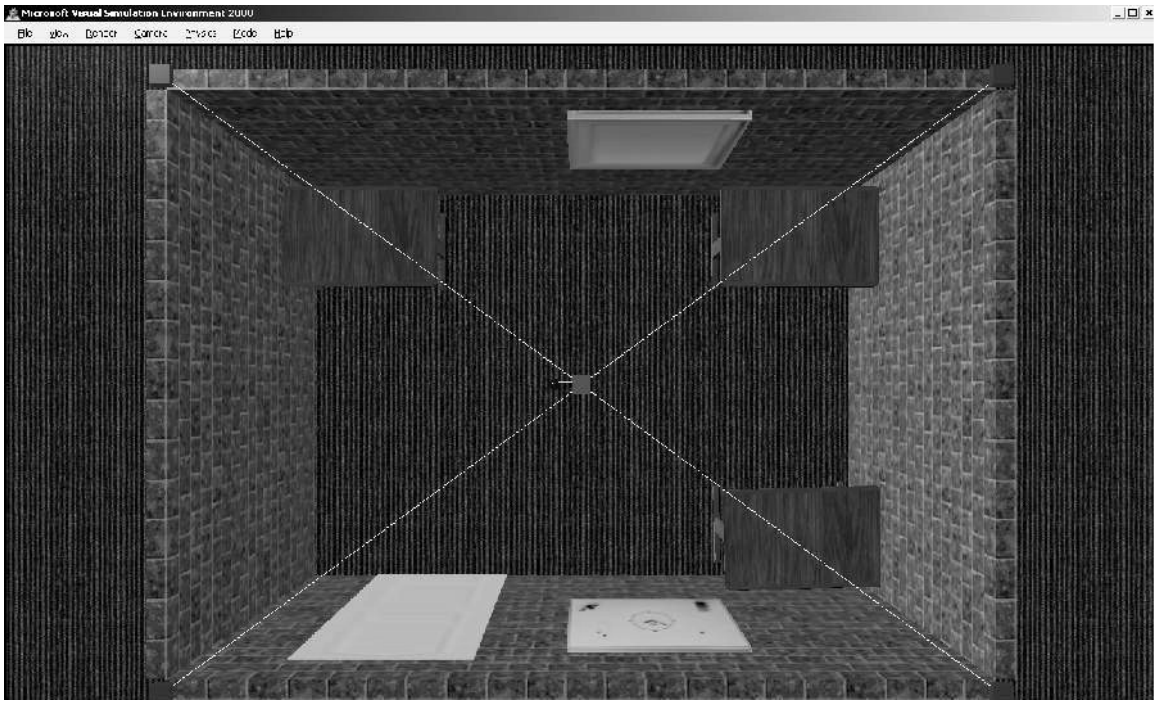


Figure 5.7. A top-down view of the simulated room. The flying part of the robot, the Flying-Base, is seen in the middle of the room with four wires connecting it to four servos winches

The main part of the simulated WireCam is its platform, also referred to as the Flying-Base, the part that the pan-tilt camera is mounted to. In this project lines are drawn to represent the wires, as no entity for wire is available in Microsoft Visual Simulation Environment. Therefore, nothing could hold the Flying-Base against the force of gravity. As a workaround solution, the Flying-Base entity is defined as a static entity. Gravity and other kind of forces do not impact on a static entity and that allows setting the position of this entity programmatically, based on the inputs and the algorithm explained in the previous section. Setting of the positioning of the Flying-Base will be explained in more details later under the section of implementation of MRDS services.

## 5.3 HARDWARE DESIGN

As illustrated in Figure 5.8, the architecture of the WireCam is the same as what was explained in the previous chapter for the pan-tilt camera with an additional part for motor controller boards and DC Servo Motors. So now the Onsite PC, in addition to the pan-tilt camera, also communicates with the servo motors through a wired connection.

The WireCam uses PIC-SERVO SC controllers, designed by J R Kerr Automation Engineering, to drive four DC servo motors. This controller is based on a PIC18F2331 which is a single chip solution for control of servo motors with incremental encoder feedback. Control of several PIC-SERVO SC controller boards is made possible by using a RS485 serial interface. This interface allows up to 32 control modules to communicate from a single serial port. In order to connect the network of controllers to a PC, as shown in Figure 5.8, a RS232-RS485 convertor (Z232-485) is used (Kerr, PIC-SERVO SC Motion Control Board).

The PIC_SERVO SC controller has a built in Network Modular Control (NMC) Communications protocol which manages the communication with all PIC-SERVO SC modules and the PC. There are a dedicated pair of transmit wires to send commands from the PC to the controller boards and a pair of wires to receive their feedback. Communications with the boards are based on their assigned addresses which are assigned to them automatically upon the initialization stage (Kerr, PIC-SERVO SC Motion Control Board). In this network, the last controller board in the line gets address 0, the board marked as A in Figure 5.8, assigned to and the addresses are assigned incrementally to the rest of the boards with the highest assigned to the very first board, board D in Figure 5.8.

This controller works in five operating modes but raw PWM output mode is the only one used in this project.

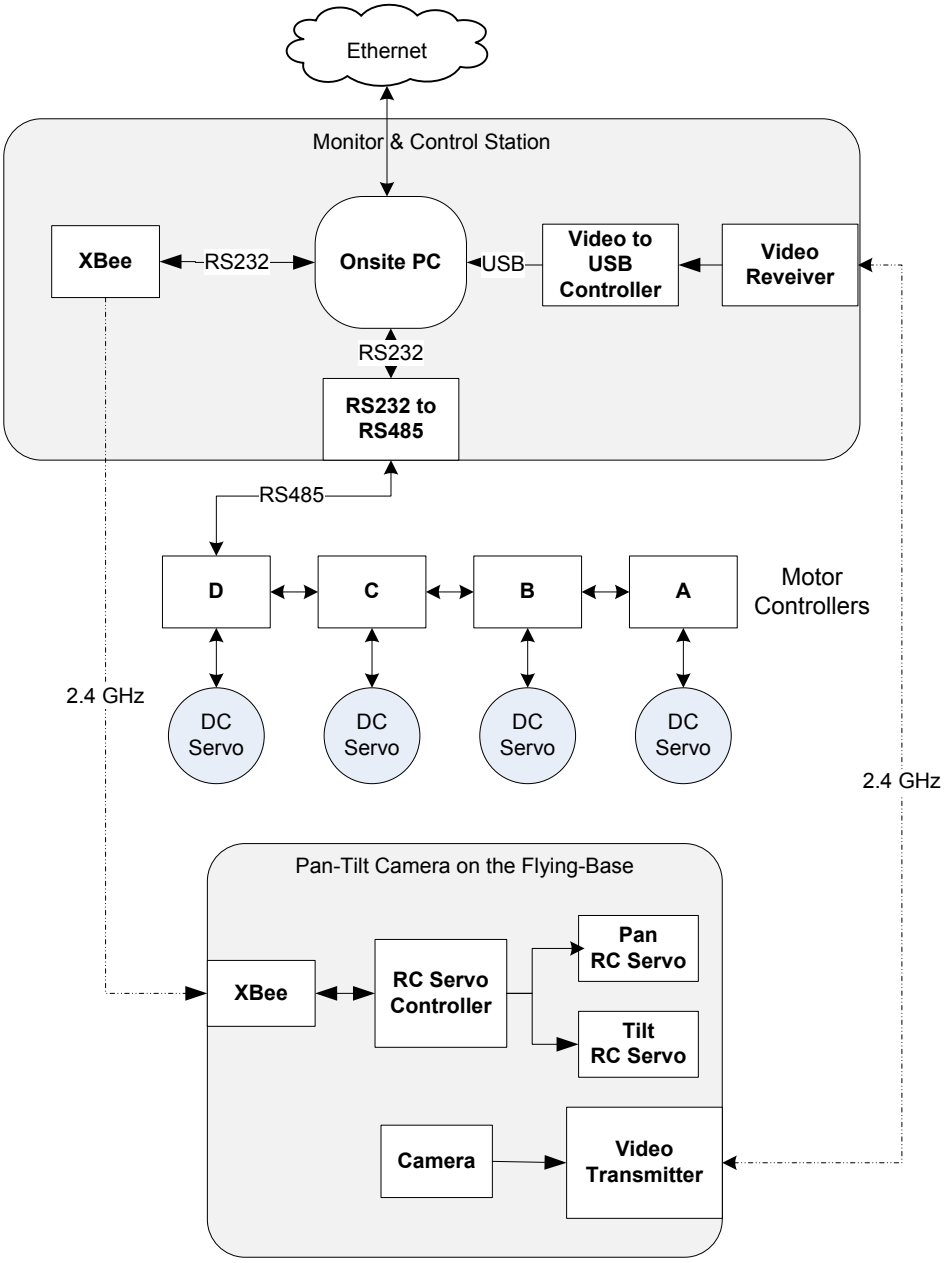


Figure 5.8. Schematics of the hardware design of the WireCam

Unlike the SSC-32 board from the previous chapter, there is no MRDS service for PIC-Servo controller boards. Using the libraries provided by J R Kerr Automation Engineering, an MRDS service was developed from scratch to make it possible to communicate with PIC-Servo controller boards using MRDS.

This MRDS service, `ServoNetwork`, takes care of initialization of the network of controllers, setting servos to their initial positions and speeds. Initialization of servo motors is another initialization performed by this service. The service sets the servo gains: Kp, Kd, and Ki. These gains are used in PID (proportional-integral-derivative) filter, which is the control filter used by the PIC-SERVO, to optimize the performance of particular motor used in this robotics system. The service also allows other MRDS services to rotate every servo in clockwise or counter clockwise directions with a variable velocity. It also manages to move all servos at a time to their preset position or velocity.

During the initialization step, first the Serial connection is established then the network of PIC-Servo controllers is initialized. At this stage addresses are assigned to each controller and the number of controllers in the network is counted.

## 5.4 INSTALLATION OF THE SYSTEM

### 5.4.1 Linear Winches

The most vital part of the hardware for this robotics system are its winches. The winch designed for this system is a linear drum type winch. Each winch comprises a DC servo motor, which includes an incremental shaft encoder, a threaded rod, a threaded drum, a drawer slide, and a fixed guide (Figure 5.9). The rod is attached to the shaft of the servo at one end and to the drum at the other end. The rod and drum are machined with the same pitch. A fixed threaded block through which the rod passes is attached to the base of the winch and thus the fixed half of the drawer slide. The servo, rod, and drum assembly is mounted to the movable half of the drawer slide. The wire is wrapped around the drum and as the rod and drum spin, the assembly slides back and forth and thus pays out or draws in the wire through the fixed guide.

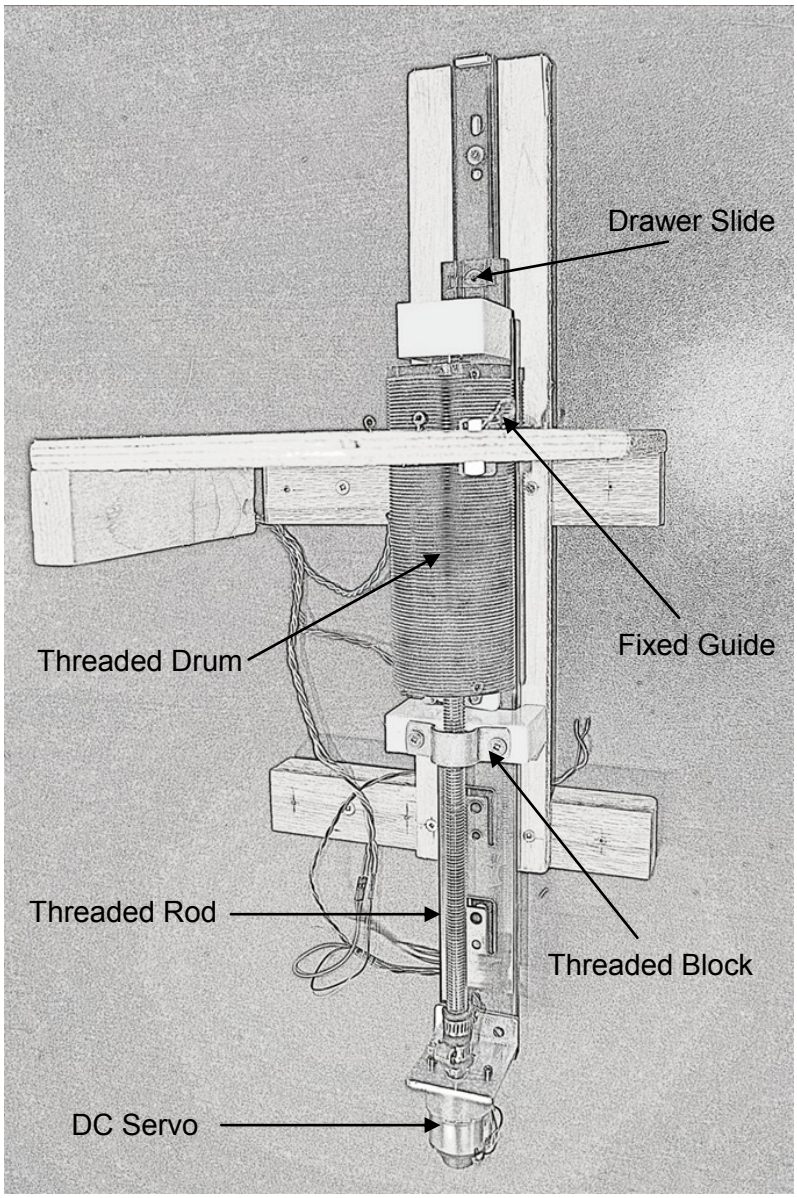


Figure 5.9. Prototype of the linear winch used in this robotics system

In this special design, the ratio of the servo's rotation to linear movement of the wire is constant. As a result, the length of wire moved per each revolution of the servo stays the same, independent to the angular position of the servo.

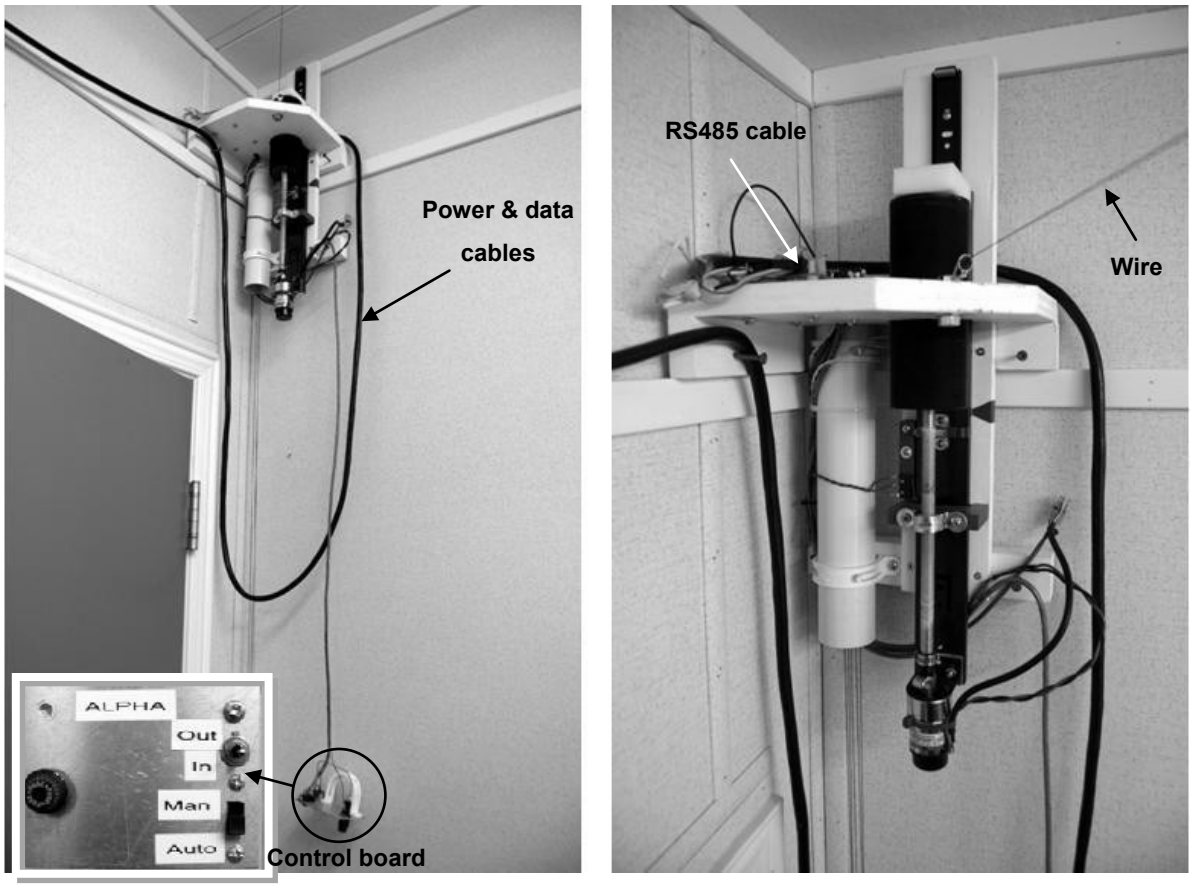


Figure 5.10. Mounted winch, its controller board, and the power & data cables

The pictures above illustrate the setup of one of the winches (Alpha) at the top corner of the Onsite room. Each winch has a control board to allow manual operation of the winch. The controller has to be switched to manual (Man) first, then using the toggle switch, labeled "Out" and "In", an operator can manually wind in and out the wire.

### 5.4.2 Electronics boards

As mentioned in the previous section, the DC servo used in each winch is controlled by a PIC-Servo SC controller board. This board is installed at the top of the winch and connects to the PC, and other motor controllers, by a RS485 cable (Figure 5.11).


Figure 5.11. PIC-Servo SC motor controller at the top of a winch

The other electronics board, the Z232-485, converts the serial port RS232 standard, connected to the OnSite PC, to RS485 standard, which is connected to the motor controllers. Figure 5.12 shows how the connections are made for power and data. The black cable carries power for the DC servos and the controller boards, and data for the controller board. A power supply provides 5 Vdc for the motor controllers and 12 Vdc for the DC motors. The commands are received from the Onsite PC through the RS232 cable and sent to the motor controllers after conversion to the RS485 standard.

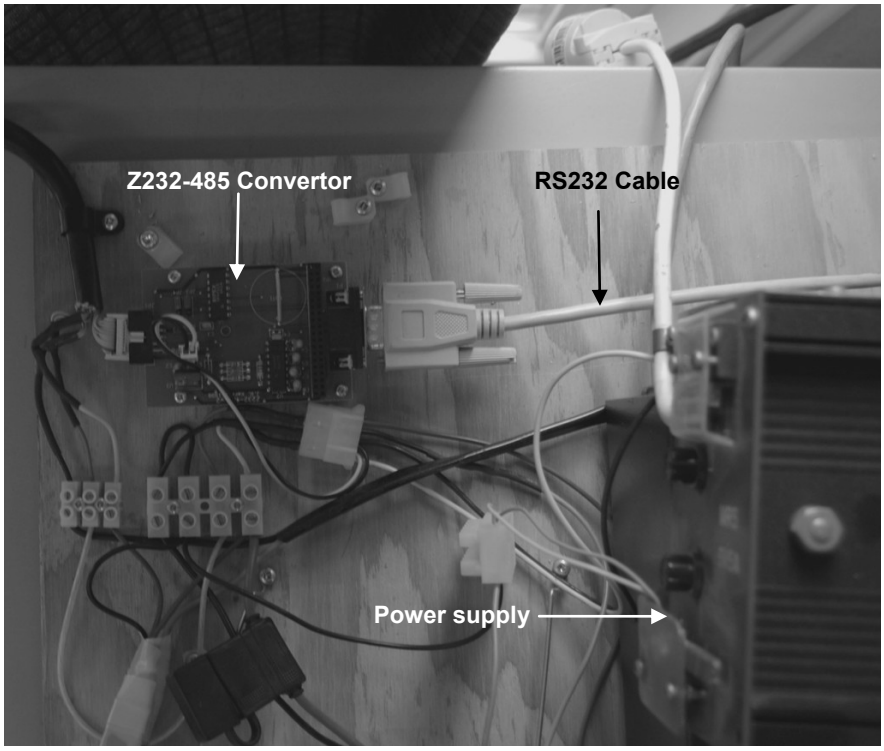


Figure 5.12. Connections of data and power cables

5.4.3 Overall view of the actual WireCam

The overall view of the robotics system (aka. WireCam) is shown in the Figure 5.13. The power and data cable is first connected to D winch and from there to C, B, and A winches in series. The Flying-Base is located in the middle of the room and is connected to each of the winches through a wire. As explained in chapter 4, the pan-tilt camera, mounted to the Flying-Base, is a wireless system and no wire is connected to it for transmission of data. A 7.2 Vdc battery is used to power the pan-tilt motors, the control board, and the Xbee module.

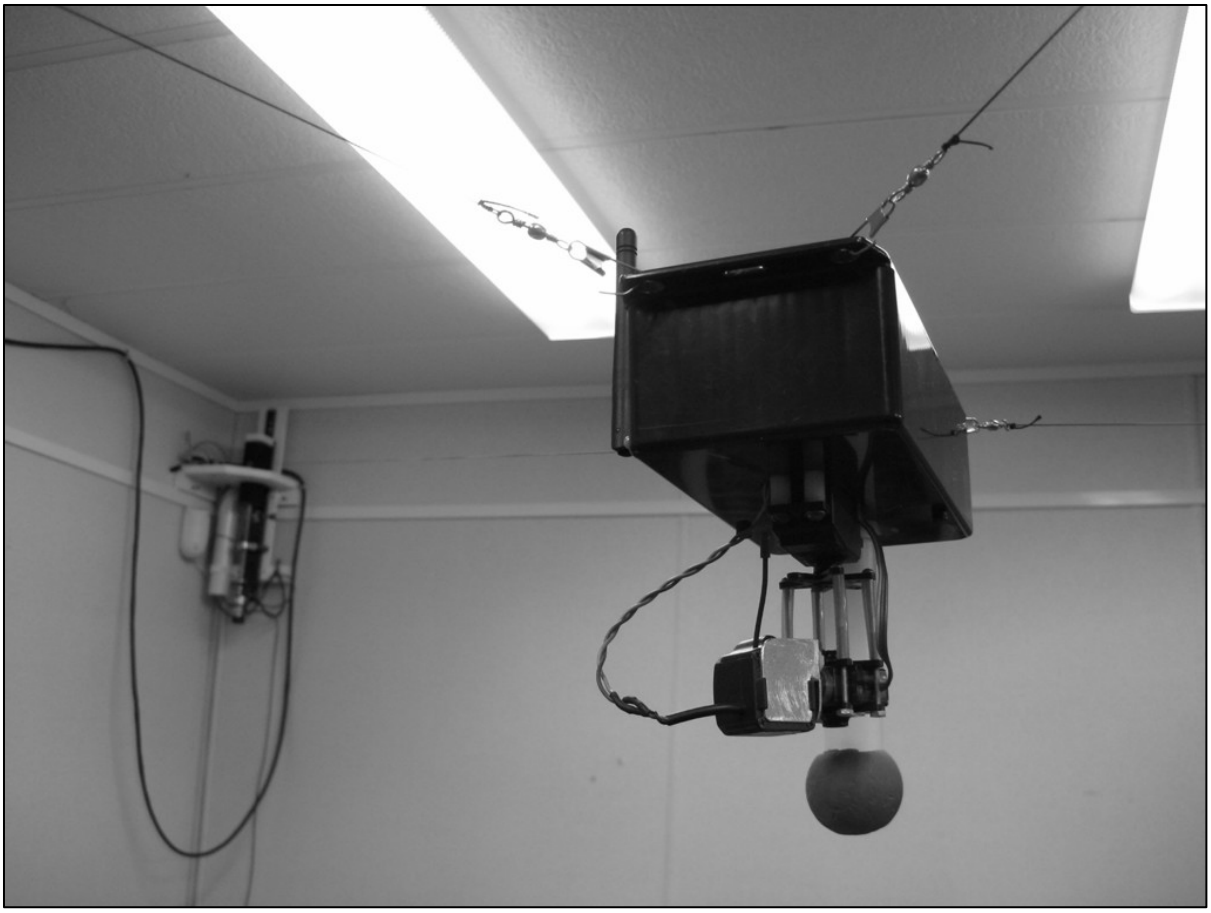

Figure 5.13. The actual WireCam with the Flying-Base in the middle and one of the winches in the left background

## 5.5 Implementation of Services And Execution of the Program

Once the simulated model and the actual system were ready it was time to run them in MRDS. The following block diagram, Figure 5.14, is an overall view of MRDS services used in this system and their relationships.

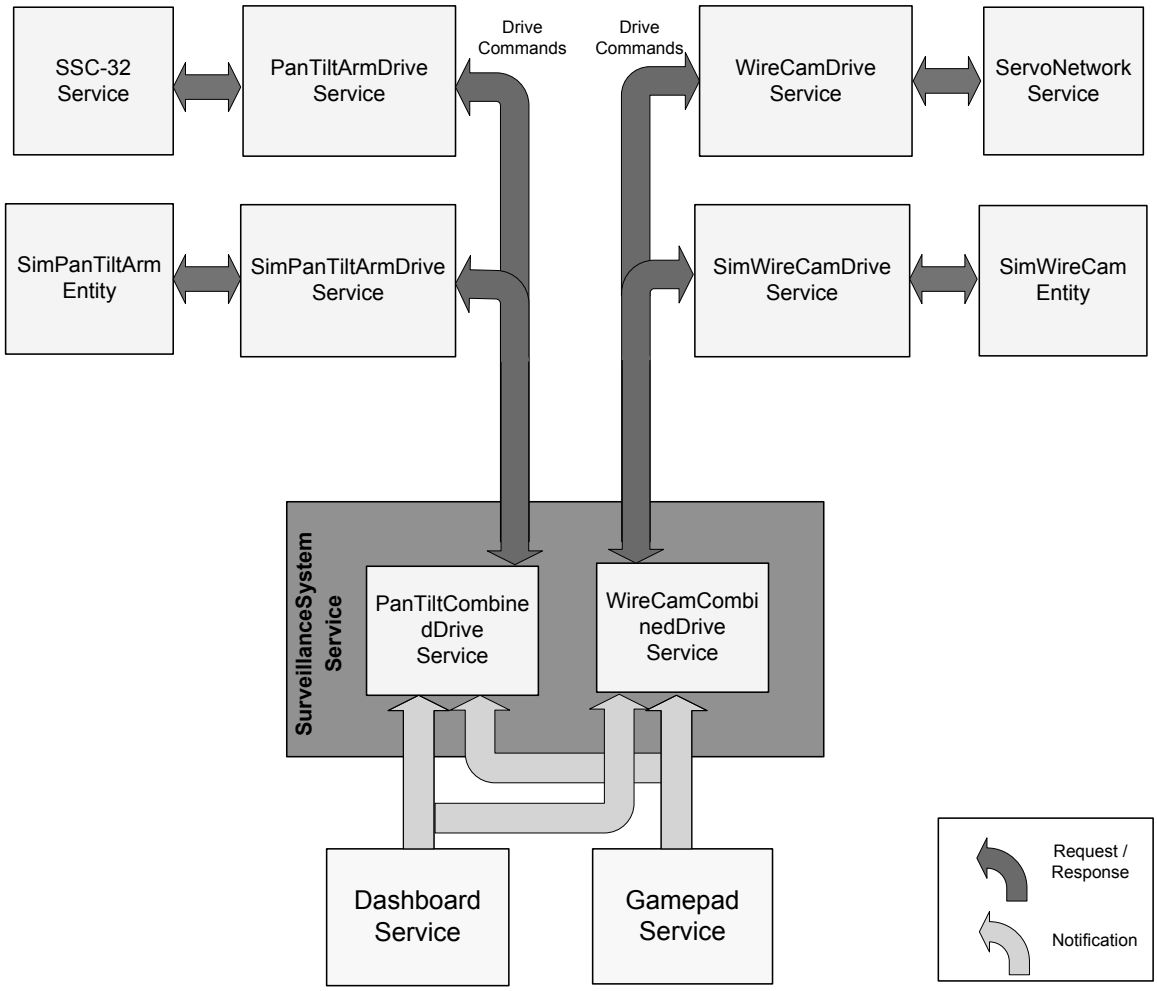


Figure 5.14. Schematic of the full system services, including the Pan-tilt camera services.

As can be seen in the figure above, all services from the chapter 4 (Pan-Tilt Camera services) are used in the WireCam. Five new services are added to the system to control the servos. `WireCamDrive` service is a service to control the WireCam. It communicates to the motor controller board through `ServoNetwork` service, which is an MRDS wrapper to the NMC Modules Dynamic Link Library (DLL). NMC Modules DLL is a Windows DLL provided by the manufacture of the controller boards and includes functions for communicating with the PIC-SERVO, PIC-STEP and PIC-I/O modules. In this project only PIC-SERVO module is used.

`WireCamDrive` service is an implementation of the Generic WireCam Drive. Similar to Pan-Tilt Camera Drive Service, extending drive services from a generic service allows us to communicate with multiple implementations of the generic service; a feature that was used in

this project to move the WireCam robot and its simulated model synchronously. The Generic WireCam Drive service was created from scratch and its support logics were then added to the Dashboard service.

`SimWireCamDrive` service is also an implementation of the Generic WireCam Drive service and controls the simulated WireCam entity. Both drive services receive posted requests from `WireCamCombinedDrive` service, which is a repeater service that receives inputs from the gamepad or dashboard services. The functionality of this service is the same as PanTiltCombinedDrive, explained in detail in chapter 4. The last service is `SurveillanceSystem` service. It is an orchestration service that calls all services required to run the system. It loads the Dashboard, `PanTiltCombinedDrive`, and `WireCamcombinedDrive` services and the other services are loaded by these three services.

### 5.5.1 Simulated WireCam Drive Service

Although the Simulated WireCam Drive (`SimWireCamDrive`) service uses the same algorithm as the WireCam Drive, there is one step further required in the simulated drive service: calculating the speed of the Flying-Base. As explained earlier, there is no entity for wire in Microsoft Visual Simulation Environment and therefore the position of the simulated Flying-Base does not change based on the tension of wires and spin of servos. Servos and wires in this simulated model are not really functional but they are there to make the simulated model look more realistic. However, the servo properties such as their velocities and encoder counts are the same in both drive services. These properties are used in the simulated drive service to find the real-time position of the Flying-Base.

Code snippet 5-1 SimWireCamDrive.SetFlyingBaseVelocityHandler method

```
public IEnumerator<ITask> SetFlyingBaseVelocityHandler(drive. SetFlyingBaseVelocity setVelocity)
{
   if (_entity == null)
      throw new InvalidOperationException("Simulation entity not registered with service");


   if (_entity != null & _entity.servoA !=null)
   {
                  float delta_z;
```

```
                float x_now = _entity.Position.X;
                float y_now = _entity.Position.Y;
                float z_now = _entity.Position.Z;

                double to_P0 = Math.Sqrt(Math.Pow(((CONSTANTS.Xmax / 2.0)), 2) +
Math.Pow(((CONSTANTS.Ymax / 2.0)), 2));


                // Z speed is controlled in another method
                delta_z = (float)CONSTANTS.Z_MAX * _entity.flyingBase.Speed_z;

                // Find the next coordination: P'(x,y,z)
                float x_next = (float)(((setVelocity.Body.X * CONSTANTS.X_MAX) /
(double)CONSTANTS.X_INPUT_MAX) + x_now);
                float y_next = (float)(((setVelocity.Body.Y * CONSTANTS.Y_MAX) /
(double)CONSTANTS.Y_INPUT_MAX) + y_now);
                float z_next = z_now + delta_z;

                //Current Distances
                double[] distances_now = Kinematics.inverse(x_now, y_now, z_now);

                //Next Distances
                double[] distances_next = Kinematics.inverse(x_next, y_next, z_next);

                // Find the current velocity of each servo
                _entity.servoA.Velocity = (float)((distances_next[0] - distances_now[0]) *
CONSTANTS.COUNTS_per_METER / CONSTANTS.T_default);
                _entity.servoB.Velocity = (float)((distances_next[1] - distances_now[1]) *
CONSTANTS.COUNTS_per_METER / CONSTANTS.T_default);
                _entity.servoC.Velocity = (float)((distances_next[2] - distances_now[2]) *
CONSTANTS.COUNTS_per_METER / CONSTANTS.T_default);
                _entity.servoD.Velocity = (float)((distances_next[3] - distances_now[3]) *
CONSTANTS.COUNTS_per_METER / CONSTANTS.T_default);

                // Find the current Count of each servo
                _entity.servoA.Encoder_count = (float)((distances_now[0] - to_P0) *
CONSTANTS.COUNTS_per_METER);
                _entity.servoB.Encoder_count = (float)((distances_now[1] - to_P0) *
CONSTANTS.COUNTS_per_METER);
                _entity.servoC.Encoder_count = (float)((distances_now[2] - to_P0) *
CONSTANTS.COUNTS_per_METER);
                _entity.servoD.Encoder_count = (float)((distances_now[3] - to_P0) *
CONSTANTS.COUNTS_per_METER);

                // Find the velocity of the P on each Axis
```

```
                _entity.flyingBase.Speed_x = (float)((setVelocity.Body.X * CONSTANTS.X_MAX) /
((double)CONSTANTS.X_INPUT_MAX * CONSTANTS.T_default));
                // multiplied by -1 to match the motion direction with the simulator's coordination
system
                _entity.flyingBase.Speed_y = -1f*(float)((setVelocity.Body.Y * CONSTANTS.Y_MAX) /
((double)CONSTANTS.Y_INPUT_MAX * CONSTANTS.T_default));

  }
        setVelocity.ResponsePort.Post(DefaultUpdateResponseType.Instance);

        // send update notification for entire state
        _subMgrPort.Post(new submgr.Submit(_state, DsspActions.UpdateRequest));
        yield break;
}
```

In the above code from the `SimWireCamDrive` service, _entity is a reference to the WireCam entity. The purpose of this method is to find the new velocity of the Flying-Base in X and Y directions, based on the algorithm explained in the previous sections, and set it in the entity. These numbers will then be used in the Update method of the Flying-Base entity to render it in the simulated scene at the right position.

Code snippet 5-2 FlyingBase.Update method

```
public override void Update(FrameUpdate update)
{
  base.Update(update);

  // Update the position of the Flying-Base based on the current speed
  xna.Vector3 new_pose;
  new_pose.X = this.State.Pose.Position.X + (float)(Speed_x * update.ElapsedTime);
  new_pose.Y = this.State.Pose.Position.Y + (float)(Speed_y * update.ElapsedTime);
  new_pose.Z = this.State.Pose.Position.Z + (float)(Speed_z * update.ElapsedTime);

  PhysicsEntity.SetPose(new Pose(TypeConversion.FromXNA(new_pose)));
}
```

### 5.5.2 WireCam Drive Service

In the WireCam Drive Service the algorithm used to calculate the position of the Flying-Base is the same as what was used in the simulated drive without the additional step of finding the velocity of the Flying-base. The PIC-Servo SC controller module has several

layers of control modes of operation (Figure 5.15) and based on requirements it can operate in any of these layers (Kerr, PIC-SERVO Motion Control). The velocity mode is the operation mode selected for this application. In this mode, as the input changes, the velocity of servos increases or decreases slowly, which creates smooth motions on the servos.

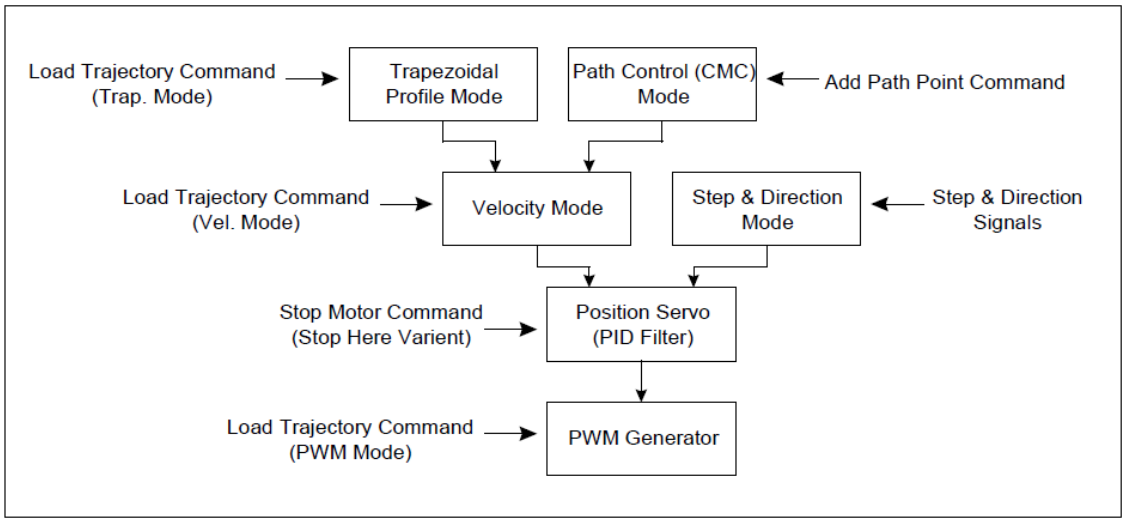


Figure 5.15. Control layers of PIC-Servo SC (Kerr, PIC-SERVO Motion Control)

To run in Velocity Mode, the trajectory of each servo has to be first loaded with the value of the velocity and the Start_Now flag, basically makes the servo start to move right away.

Code snippet 5-3 ServoLib.SetServoTraj method

```
/// <summary>
/// Set trajectory for a servo but no move until GroupMove is called
/// </summary>
/// <param name="addr">Address of servo</param>
/// <param name="counterClockwise">If false then direction=clockwise else, other direction</param>
/// <param name="vel">servo velocity</param>
public static void SetServoTraj(byte addr, bool clockwise, int vel)
{
  byte mode;
  int pos, acc;
  byte pwm;
  bool result;

  byte current_status;
```

```
  int move_complete = 0;

  mode = (byte)(Constants.LOAD_VEL | Constants.LOAD_ACC | Constants.ENABLE_SERVO |
Constants.VEL_MODE | Constants.START_NOW);
  pos = 0;
  acc = 100;
  pwm = 0;

  NMCLIB.ServoLoadTraj(addr, mode, pos, vel, acc, pwm); // start the motion
  if (!result)
  {
    Console.Out.WriteLine("Failed to load trajectory to servo #" + addr);
  }

        int counter = 0;
        //wait for the status
        while (move_complete == 0) //wait for MOVE_DONE bit to go HIGH
        {
          NMCLIB.NmcNoOp(addr); //poll controller to get current status data
          current_status = NMCLIB.NmcGetStat(addr);
          move_complete = current_status & Constants.MOVE_DONE;

          // Don't wait forever. Eventually we have to give up!
          counter++;
          if (counter > 30)
          {
            Console.Out.WriteLine("MOVE_DONE hasn't been set to HIGH in servo #" + addr);
            break;
          }
        }
}
```

The algorithm of WireCam drive is implemented as follow. First, the current position of the Flying-Base is calculated based on its distance from each servo. Second, the desired direction of each servo is determined. Now, based on the received input the destination position and the new distances to each servo are calculated. The next step is to determine the velocity of each servo and their directions. Once all is determined, the new velocities and directions of all servos are loaded, using `SetServoTraj` method, to the control board and a group move command is sent out to make all servos update their velocities. The process waits at this state and does not accept any new input until the group move is completed.

Code snippet 5-4 ServoLib.GroupMove method

```
public static void GroupMove()
{
    NMCLIB.NmcSynchOutput(0xFF, 0x00);
}
```

The two methods mentioned above are from `ServoLib` class. Inside each of these methods `NMCLIB's` methods are called. `NMCLIB` class makes NMC DLL's functions available to `ServoLib` class. In general, a managed code cannot call functionalities outside of the .NET framework. DLLs are one example of unmanaged codes that are usually written in C++ language. `DllImport` is a solution provided in .NET to solve this problem (Roof & Fergus, 2003). In the `NMCLIB` class this solution is implemented to call the methods of the NMC Modules DLL (`NMCLIB04.dll`). The following example from `NMCLIB` class shows how `ServoLoadTraj` method from `NMCLIB04` library is imported to the C# code.

Code snippet 5-5 NMCLIB.ServoLoadTraj method

```
[DllImport("NMCLIB04.dll", EntryPoint = "ServoLoadTraj", CharSet = CharSet.Unicode)]
public static extern bool ServoLoadTraj(byte addr, byte mode, int pos, int vel, int acc, byte pwm);
```

The following image (Figure 5.16) is a screenshot of the Operator PC screen. The operator can see the stream of the actual webcam received from the Onsite PC, where the WireCam system is installed, along with the video stream from the simulated scene. In this screenshot the entrance door of the Onsite and virtual rooms and part of the mounted whiteboard are captured in both webcams.

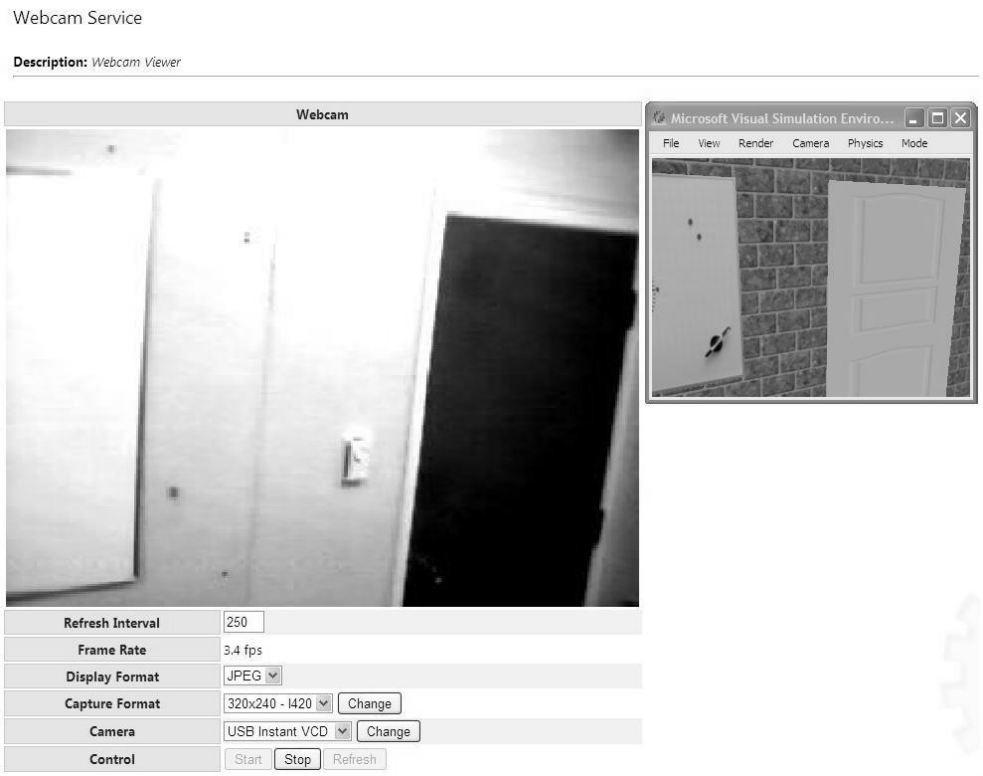


Figure 5.16. A screenshot of Operator PC during remote control of the WireCam

## 5.6 CHALLENGES DURING IMPLEMENTATION AND EXECUTION

Several issues and challenges arose during the implementation and testing of this chapter's application causing an unexpected delay in completion of the project. One of the major challenges was declaration of a pointer in C# which was required in order to work with the manufacturer's provided DLL library, `NMCLIB04.dll`.

The PIC-SERVO SC Motion Control Board comes with an open source software package and documentations. The DLL provided for this board was developed in C++ which lets developers get the DC motor up and running by calling the package's methods instead of sending commands directly to the board.

The initialization method of the C++ DLL, `NmcInit`, requires a parameter as pointer data type which is not a standard data type in C#. After several days of unsuccessful search for a workaround the decision was made to develop the C++ DLL package in C# from scratch. That attempt required detailed study of the datasheet and documentations of the board. After

some progress was made on the implementation of the DLL functions in C#, a solution was found to use pointers in C# and the implementation of the package in C# was no longer required.

Code snippet 5-6 NMCLIB04.dll's function using a pointer

```
//-------------------------------------------------------------------------------------------------//
//  Function Name:  NmcInit                                                                        //
//  Return Value:   Returns the number of controller found on network                              //
//  Parameters:     portname: name of COM port ("COMn:", where n=1-8)                              //
//                  baudrate: 19200, 57600, 115200, 230400 (PIC-SERVO only)                        //
//  Description:    Initialize the network of controllers with sequential                          //
//                  addresses starting at 1.                                                       //
//-------------------------------------------------------------------------------------------------//
extern "C" WINAPI __declspec(dllexport) int NmcInit(char *portname, unsigned int baudrate)
```

Code snippet 5-7 StringBuilder is the solution for pointer used in NMCLIB.NmcLib

```
//Initialization and shutdown
[DllImport("NMCLIB04.dll", EntryPoint = "NmcInit", CharSet = CharSet.Ansi)]
public static extern int NmcInit(StringBuilder portname, int baudrate);
```

In the design of the simulator, a couple of challenges arose. The first was the fact that there is not a standard solution to define an entity for wire. Searching for an alternative solution was finally finished by excluding the wire and the dependency of the Flying-Base to it and instead drawing lines only for illustration to the operator. Again drawing a line in the Microsoft Visual Simulation Environment is not supported and extra coding was required to make that happen.

For this chapter more coding was required as there were services required to be developed from scratch. Also similar to the last chapter it was required to alter the open source codes of MRDS. Some of the codes are complicated and in order to add a new functionality to them it is first required to study and analyse them. This process was quite time consuming and it involved a lot of debugging.

# CHAPTER 6

# CONCLUSION

The objective of this research was to review popular RDEs, select one, and try out its features for the robotics world in a practical robotics project. Microsoft Robotics Development Studio (MRDS) was selected for this research and step by step its capabilities were tried toward the goal which was implementation and demonstration of a wire-suspended surveillance system. The robot was first designed, implemented and tested in the Microsoft Visual Simulation Environment and was then built and simultaneously demonstrated successfully with its simulated model.

The experience proved that MRDS is a powerful RDE with capability to provide a lot of solutions. However, MRDS has a long learning curve and the complexity of its advanced features impacts its helpfulness in speeding up the accomplishment of a project. This robotics development environment has a wide range of features to learn which makes it hard to master and refreshing one's mind on its subjects is required from time to time.

Working with MRDS is a lot easier if the project could be limited to what this environment offers out of the box. Fewer challenges would be faced if the project goals could be accomplished by using the existing MRDS services and the Visual Programming Language (VPL). The services and simulated entities that come with MRDS are still limited. Not many projects could be completed without a need to implement custom services.

Implementation of some of the fundamental solutions in MRDS is not straightforward. Two examples of what was experienced in this project are the distribution of services and synchronized control of the actual robot and its simulated model. Both are not supported in the MRDS's standard dashboard service and are not covered in tutorials and documentation of this framework.

There are things that can be improved or changed for future work on this project:

1. In the current implementation both Pan-tilt and WireCam systems cannot be controlled at the same time due to a limitation in the design of the standard dashboard as mentioned above. In the current design only one drive service can be selected at a time. This was an issue during the implementation of each of the above systems for controlling the actual system and the simulated model of a system but it was resolved by introducing a repeater service (Combined Drive Services). However, in this additional unsolved challenge, the drive services of both systems are already extended different generic services so it is not possible to apply the same solution here. A possible solution would involve a new service that partners with both Combined Drive Services and listens to request types of both generic services and upon receiving one of them posts it to the right generic drive port. This service should have its own contract identification which is different from other generic services and recognition of it will be added to the Dashboard service. It should also cover all operations of both generic services.

2. Instead of the inexpensive wireless camera used in this project a better quality wireless IP camera could be used. IP cameras were expensive when this project first started but now with the improvement of the technology there are many inexpensive IP cameras available. One can even build a unit with Raspberry Pi for a cost of around $100 (Buenger, 2014).

3. As a recommendation for a future work, using one of the available 3D reconstruction applications for building 3D objects from the 2D photos of the object (e.g. Eos Systems Inc's PhotoModeler Scanner, and AgiSoft's PhotoScan), the simulated scene could have been built automatically. The idea is to let the WireCam scan the room as an initiation task right after installation in a room and create the simulated scene based on the images it takes. This would eliminate the need for measuring the room and programmatically adding objects to the scene.

4. It would also be advantageous to have an algorithm added to the current application to avoid collisions. Due to the delay and unexpected errors, images

from the scene might not reach the operator, and even with the simulator in place collisions might occur. However, by adding intelligence to the service to avoid collisions with objects it will be guaranteed that network issues and operator mistakes could not cause a collision.

# BIBLIOGRAPHY


Biggs, G., Rusu, R. B., Collett, T., Brian, G., & Vaughan, R. (2012). And all the robots merely Players. *IEEE Robotics & Automation Magazine* .

Buenger, C. (2014, May 11). *Raspberry Pi as low-cost HD surveillance camera*. Retrieved August 5, 2014, from Code Project: http://www.codeproject.com/Articles/665518/Raspberry-Pi-as-low-cost-HD-surveillance-camera

Cherry, S. (2007, Aug). Robots, Incorporated. *IEEE Spectrum* , 24 - 29.

D. Salle, M. T. (2007). Using Microsoft Robotics Studio for the design of generic robotic controllers: the robuBOX Software.

Das, B. and Kozey, J. (1994). A computerized potentiometric system for structural and functional anthropometric measurements. *Ergonomics , 37* (6), 1031.

Digi International Inc. (2014). *XBee® ZB RF modules utilizing the ZigBee PRO Feature Set*. Retrieved 04 18, 2014, from Digi: http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/zigbee-mesh-module/xbee-zb-module#overview

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.*

Gates, B. (2007). A robot in every home. *Scientific American , 296* (1), 58-65.

Gerkey, B. P., Vaughan, R. T., & Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems. *In Proceedings of the 11th international conference on advanced robotics*, *1*, pp. 317-323. Coimbra,Portugal.

*Google is to start building its own self-driving cars.* (2014, May 27). Retrieved September 19, 2014, from BBC News: http://www.bbc.com/news/technology-27587558

Griffin, T. (2010). The Art of LEGO MINDSTORMS NXT-G Programming. (pp. 5-6). No Starch Press.

Jackson, J. (2007). Microsoft robotics studio: A technical introduction. *IEEE robotics automation magazine* , 82.

Johns, K., & Taylor, T. (2008). *Professional Microsoft Robotics Developer Studio.* Wrox.

Kerr, J. (n.d.). *PIC-SERVO Motion Control.* Retrieved 07 25, 2014, from PIC-SERVO Motion Control, Jeffery Kerr, LLC: www.jrkerr.com/picsrvsc.pdf

Kerr, J. (n.d.). *PIC-SERVO SC Motion Control Board.* Retrieved Mar 23, 2012, from PIC-SERVO Motion Control: http://www.jrkerr.com/pssc_bd.pdf

Koenig, N., & Howard, A. (2004). Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. *Proceedings 01 2004 IEEE!RSJ International Conference on Intelligent Robots and Systems.* Sendai, Japan.

*Korean baseball team fills stands with 'Fanbots'.* (2014, July 29). Retrieved September 19, 2014, from CTV News: http://www.ctvnews.ca/sci-tech/korean-baseball-team-fills-stands-with-fanbots-1.1936686

Kramer, J., & Scheutz, M. (2007). Development Environments for Mobile Autonomous Robots: A Survey. *Autonomous Robots , 22*, 101-132.

Lynxmotion. (2007, Jan 31). *SSC-32 Ver 2.0.* Retrieved Mar 20, 2012, from http://www.lynxmotion.com/images/data/ssc-32.pdf

McComb, G. (2011). *The Robot Builder's Bonanza.* New York: McGraw-Hill/TAB Electronics.

Microsoft. (2008, Nov 17). *Microsoft Unveils Microsoft Robotics Developer Studio 2008*. Retrieved Apr 9, 2012, from Microsoft News Center: http://www.microsoft.com/presspass/press/2008/nov08/11-17robodevelopmentpr.mspx

Microsoft. (2012). *Robotics Tutorial 4 (VPL) - Drive-By-Wire.* Retrieved Feb 10, 2014, from MSDN: http://msdn.microsoft.com/en-us/library/bb483038.aspx

Microsoft. (2012). *Welcome to Robotics Developer Studio*. Retrieved Feb 10, 2014, from MSDN: http://msdn.microsoft.com/en-ca/library/bb648760.aspx

Oliveira, L. B. and Osório, F. S. and Nakagawa, E. Y. (2012). *A Systematic Review on Service-Oriented Robotic Systems Development.* University of São Paulo.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). ROS: an open-source Robot Operating System. *ICRA Workshop on Open Source Software , 3* (3.2).

Roof, L., & Fergus, D. (2003). Working with Unmanaged Code. In *The Definitive Guide to the .NET Compact Framework* (pp. 689-734). Apress.

*ROS Tutorials*. (2013, 09 28). Retrieved Feb 21, 2014, from ROS Wiki: http://www.ros.org/

*Russian pizza chain claims to deliver by drone.* (2014, June 23). Retrieved September 19, 2014, from CBC News: http://www.cbc.ca/news/russian-pizza-chain-claims-to-deliver-by-drone-1.2684763

*SAFAR*. (2012). Retrieved Jan 22, 2012, from MobotSoft: http://www.mobotsoft.com/?page_id=82

Staranowicz, A., & Mariottini, G. L. (2011). A survey and comparison of commercial and open-source robotic simulator software. *in Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments (PETRA'11).* Crete: ACM.

Weiss, R., & Overcast, I. (2008). Finding your bot-mate: criteria for evaluating robot kits for use in undergraduate computer science education. *Journal of Computing Sciences in Colleges , 24* (2), 43-49.

Zoppi, A. (2012). *A Lightweight Open-source Communication Framework for Native Integration of Resource-onstrained Robotics Devices with ROS.* Milan: Politecnico di Milano.