**RoboWars Final Report**

By  
Alexander Craig  
Alexander Dinardo  
Steve Legere  
Michael Wright

Supervisor: Cheryll Schramm, P.Eng

A report submitted in partial fulfillment of the requirements  
of SYSC 4907 Engineering Project

Department of Systems and Computer Engineering  
Faculty of Engineering  
Carleton University

April 5th, 2011

**Abstract**

Test

# Table of Contents

[1 Table of Contents 1](#_Toc289688303)

[2 List of Figures 3](#_Toc289688304)

[3 List of Tables 3](#_Toc289688305)

[4 Introduction 4](#_Toc289688306)

[5 The Engineering Project 7](#_Toc289688307)

[5.1 Health and Safety 7](#_Toc289688308)

[5.2 Engineering Professionalism 8](#_Toc289688309)

[5.3 Project Management 8](#_Toc289688310)

[5.4 Individual Contributions 9](#_Toc289688311)

[5.4.1 Project Contributions 9](#_Toc289688312)

[5.4.2 Report Contributions 9](#_Toc289688313)

[6 Background and Terminology 10](#_Toc289688314)

[6.1 Android 10](#_Toc289688315)

[6.1.1 Operating System 10](#_Toc289688316)

[6.1.2 Emulator 11](#_Toc289688317)

[6.1.3 Additional Libraries 12](#_Toc289688318)

[6.2 Lego Mindstorm NXT 2.0 12](#_Toc289688319)

[6.2.1 Robotic Requirements 12](#_Toc289688320)

[6.2.2 Mindstorm NXT 2.0 Capabilities 13](#_Toc289688321)

[6.2.3 RoboWars Robot Design 14](#_Toc289688322)

[7 Requirements 14](#_Toc289688323)

[8 Overall System Architecture 20](#_Toc289688324)

[9 Server Implementation 22](#_Toc289688325)

[9.1 Controller – User and Robot State Management and Data Propagation 23](#_Toc289688326)

[9.2 Controller – Real Time Video Streaming 30](#_Toc289688327)

[9.2.1 Alternative Solution – RTP/RTSP Streaming 30](#_Toc289688328)

[9.2.2 Implemented Solution – Custom UDP Protocol 32](#_Toc289688329)

[9.3 View – Administration Panel and Camera Configuration 36](#_Toc289688330)

[9.4 View – 2D Display 39](#_Toc289688331)

[9.5 Model – Virtual World Model Implementation 40](#_Toc289688332)

[9.5.1 Architecture 40](#_Toc289688333)

[9.5.2 Use of Java Event Model 42](#_Toc289688334)

[9.5.3 GameEntities and 2D Collision Detection 42](#_Toc289688335)

[9.5.4 GameRobot and RobotCommand Priority 43](#_Toc289688336)

[9.5.5 Android Incompatibility with LeJOSPose 44](#_Toc289688337)

[9.6 Server-Side Logging 45](#_Toc289688338)

[10 Robot Client Implementation 46](#_Toc289688339)

[10.1 Client Implementation 46](#_Toc289688340)

[10.2 LeJOS and Modifications 48](#_Toc289688341)

[10.2.1 LeJOS 48](#_Toc289688342)

[10.2.2 Bluetooth Communications and I/O Modifications 48](#_Toc289688343)

[10.2.3 Navigation and Piloting Modification 50](#_Toc289688344)

[10.3 Position Tracking and Error Correction 52](#_Toc289688345)

[10.3.1 Position Tracking 52](#_Toc289688346)

[10.3.2 Error Correction and the Grid 53](#_Toc289688347)

[10.4 Future Suggestions 54](#_Toc289688348)

[11 Android Client Implementation 55](#_Toc289688349)

[11.1 Android Libraries and Application Layout 55](#_Toc289688350)

[11.1.1 Android Libraries 55](#_Toc289688351)

[11.1.2 Application Layout 56](#_Toc289688352)

[11.2 OpenGL Rendering 58](#_Toc289688353)

[11.3 Development Issues and Solutions 58](#_Toc289688354)

[11.3.1 Android Emulator 58](#_Toc289688355)

[12 Testing 59](#_Toc289688356)

[12.1 Robot Client Testing 59](#_Toc289688357)

[12.2 Android Client Testing 60](#_Toc289688358)

[12.3 Server Side Testing 61](#_Toc289688359)

[12.3.1 Controller Testing 61](#_Toc289688360)

[12.3.2 Model Testing 64](#_Toc289688361)

[12.3.3 Integration Testing 65](#_Toc289688362)

[13 Conclusion 65](#_Toc289688363)

[14 References 67](#_Toc289688364)

[Appendix A – Schedule Gantt Charts 69](#_Toc289688365)

[A.1 – Original Schedule (from Project Proposal) 70](#_Toc289688366)

[A.2 – Revised Schedule (Revised for Progress Report, Dec 6th, 2010) 71](#_Toc289688367)

[Appendix B – Use Cases 72](#_Toc289688368)

[B.1 – Actor Definitions 73](#_Toc289688369)

[B.2 – Use Case Specification 74](#_Toc289688370)

[Appendix C – Use Case Realizations (Sequence Diagrams) 83](#_Toc289688371)

[Appendix D – Class Diagrams 89](#_Toc289688372)

[D.1 – Server-Side Overall Class Diagram 90](#_Toc289688373)

[D.2 – Detailed Controller Class Diagram 91](#_Toc289688374)

[D.3 – Detailed Model Class Diagram 92](#_Toc289688375)

[D.4 – Robot Class Diagram 93](#_Toc289688376)

[D.5 – Android Client Class Diagram 94](#_Toc289688377)

# List of Figures

[Figure 7.1: RoboWars use case diagram 15](#_Toc289688392)

[Figure 8.1: A deployment diagram of the complete RoboWars system. Note that multiple instances of the Robot and Android nodes can be serviced by a single Central Server node. 20](#_Toc289688393)

[Figure 9.1: Class diagram of the main classes concerning user and robot state management in the server side controller component 23](#_Toc289688394)

[Figure 9.2: Sequence diagram of Android client registration 24](#_Toc289688395)

[Figure 9.3: Sequence diagram of robot client registration 26](#_Toc289688396)

[Figure 9.4: Collaboration diagram outlining the game launch process. 29](#_Toc289688397)

[Figure 9.5: Deployment diagram of the testing system used for RTP/RTSP streaming 31](#_Toc289688398)

[Figure 9.6: Class diagram of classes responsible for media streaming 33](#_Toc289688399)

[Figure 9.7: The administrator view of the RoboWars server lobby 36](#_Toc289688400)

[Figure 9.8: A screenshot of the camera configuration and preview GUI 38](#_Toc289688401)

[Figure 10.1: How commands are received 47](#_Toc289688402)

[Figure 10.2: RoboWarsNavigator structure 47](#_Toc289688403)

[Figure 11.1:A simple LinearLayout implementation using only XML 56](#_Toc289688404)

[Figure 11.2: XML code used to obtain the GUI as seen in figure 11.1 above 57](#_Toc289688405)

[Figure 12.1 – An example of coverage output from EclEmma 62](#_Toc289688406)

# List of Tables

[Table 7.1: RoboWars use case names and descriptions 16](#_Toc289688407)

# Introduction

This report is intended as a final report of progress by the RoboWars SYSC 4907 engineering project team. The RoboWars project aims to further develop solutions in the area of remote robotics control systems. Control systems for remote robotics available today often require custom, specialized hardware to interface with the robotics, and may provide only rudimentary control functionality. By exploiting widely available consumer hardware and open source software, the RoboWars project aims to create a feature rich robotics control system that can be easily adapted to any Bluetooth capable robotics system.

Developing a robotics control system on a widely available mobile platform provides developers with greater accessibility and reduced development time. Using smartphones to remotely control devices, robotic or not, eliminates the need for extra time and development resources to be spent on designing and building a specialised hardware platform to run the control software on. With smartphones already in widespread use by the public it is simple and intuitive for a user to download, install, and immediately use the application to remotely control their device. This provides developers with a well established and easily accessible customer base.

In addition to avoiding hardware development costs, smartphones provide a powerful platform which can support a wide range of features to enhance the usability of the controlled robotics. Smartphones are typically equipped with high resolution touch screens, Wi-Fi and Bluetooth communication capabilities, accelerometers, and dedicated graphics processors. These capabilities allow a robotics system to support enhanced features that may not be available with simpler dedicated hardware, such as live video streaming, three dimensional graphical rendering, tilt or touch screen based movement control, and high bandwidth data exchange to support advanced robot behaviour.

The RoboWars project has two primary objectives:

* The first project objective is to develop a robotics control system which is both intuitive to use and is implemented on a mobile platform that is widely available and used by the public. Specifically, the project targets mobile smartphones running the Android operating system as the client platform.
* The second project objective is to experiment with the combination of live video and virtually generated, overlaid imagery to enhance the ease of use and feature set of a robotics control system. This technology is commonly referred to as augmented reality.

To combine these objectives, the project aims to create a solution which allows two remotely controlled LEGO Mindstorm NXT 2.0 robots to share and interact with a simple virtual world which will be rendered in three dimensions and overlaid on a live video feed to be displayed to the remote operators. Users can connect to the system from smartphones anywhere where an internet connection is available, and use the robots to play simple virtual games supported by augmented reality overlays. Specifically, the project aims to support two game modes: a tank simulation, and “light-cycles.” In the tank simulation game mode players are given full analog control of their robots, and virtual projectiles are used to attack and remove other robots from the game. In the “light-cycles” game mode each player’s robot must constantly move forward, and must avoid the virtual walls generated behind each robot. In each case, the calculated position of each robot in the server’s internal state must match the actual physical location of each robot at all times for consistent game play.

The following components of the proposed solution have been successfully implemented:

* Centralized server software which manages connections, propagates commands and messages, and records client state from Android clients (via standard IP) and Mindstorm NXT 2.0 robotics kits (via Bluetooth).
* Mindstorm NXT 2.0 Intelligent Brick software which supports duplex Bluetooth communication, remote execution of commands received from the central server, dead reckoning position tracking and error correction, and real time position and heading data transmission to the central server.
* Server side virtual world implementation which is capable of asynchronously updating robot positions, updating model physics in real time, filtering client commands based on validity against the current game state, and generating outgoing robot commands to ensure physical robot movement does not violate the virtual game state.
* Real time video streaming from a USB webcam connected to the central server to Android clients using a custom UDP protocol.
* Android client software which supports OpenGL rendering of an incoming real time video stream, as well as rudimentary three dimensional graphics rendering (simple textured polygons).

The remainder of this report will detail the development methodology of the RoboWars project, and the technical specifications of the implemented solution. First, the requirements for the project will be discussed, as well as the engineering principles and development practices used to implement the proposed solution. Next, the technical details of the implementation will be discussed, followed by a discussion of the testing methodology employed and results obtained. Finally conclusions and recommendations for alternative implementations and further work will be presented.

# The Engineering Project

## Health and Safety

Due to the nature of the project, health and safety was not a major concern. The laboratories utilized at Carleton University were kept in a clean state at all times. All emergency exits were kept clear of obstructions, and all health and safety regulations, as well as laboratory rules and regulations, were adhered to. Laboratory equipment was treated respectfully at all times.

## Engineering Professionalism

All members of the RoboWars engineering project displayed professional attitude and competence when carrying out project responsibilities, and reflected engineering principles at all times. All members participated in weekly project meetings, each voicing their accomplishments since the last meeting, as well as their goals for the next meeting. The project team remained well organized and communicated all problems or setbacks with other members before coming to a unanimous solution.

## Project Management

The RoboWars project was kept very well organized throughout its duration. Two primary methods of management were invoked in order to obtain a high level of efficiency:

* GitHub: Git [1] is an open source distributed version control system which was utilized through the free service GitHub [2] in order to store the project. GitHub allowed every member to contribute to the project at any given time, with real-time updates to each member of the team. GitHub keeps track of each member's commitments to the project and allows others to view every modification or addition made to the project.
* Weekly Meetings: All four members of the RoboWars project, along with its supervisor Cheryl Schramm, convened diligently every week in order to discuss project progress, problems, and future work to be completed.

## Individual Contributions

### Project Contributions

All members participated in requirements elicitation and initial system design.

* **Alexander Craig:**
  + Server-side player and robot state management and communication paths (controller component of central server)
  + Video streaming (both server-side transmission and client-side reception of frames)
  + Client simulator and software robot simulations.
* **Alexander Dinardo:**
  + Game model implementation
  + Game design
  + 2D collision detection algorithm design
* **Steve Legere:**
  + Android client implementation and testing
* **Michael Wright:**
  + Robot side client implementation
  + Bluetooth IO implementation
  + Robot side error correction and position tracking

### Report Contributions

* **Alexander Craig:** Introduction, Background, Requirements (7), Overall System Architecture (8), Server Implementation (9.1-9.3, 9.6), Testing (12.3.1)
* **Alexander Dinardo:** Server Implementation (9.4, 9.5), Testing (12.3.2 - 12.3.3).
* **Steve Legere:** Engineering Project (5), Background (6.1), Android Client Implementation (11), Testing (12.2).
* **Michael Wright:** Abstract, Background (6.2), Robot Client Implementation (10), Testing (12.1), Conclusion (13).

# Background and Terminology

## Android

Android is a set of software which includes a mobile Operating System (OS), middleware, and required applications. Android is open source software (OSS), meaning the source code is freely available to the general public to use and modify so long as the licensing restrictions are adhered to.

### Operating System

The Android architecture is layered in four key categories: applications, framework, libraries, and kernel. An additional fifth layer, the runtime layer, contains core libraries essential in any application or program, as well as the Dalvic Virtual Machine (DVM). [3]

The Android core libraries allow the Java programming language to be seamlessly implemented and used on Android mobile devices alongside the DVM. DVM is a non-licensed implementation of the Java VM which was built for mobile devices; it is small, efficient and optimized for mobile devices such as the HTC Desire. [3]

The libraries included in the Android software development kit (SDK) are:

* Libc: Standard C library;
* SSL: Secure Socket Layer;
* SGL: Custom engine library for 2D images;
* OpenGL/ES: An engine library for 3D images;
* Media: Library which deals with all multimedia intents;
* SQLite: Deals with SQL databases, used for data storage;
* WebKit: Library for web-based applications, such as browsers;
* FreeType: Deals with images and vectors; and
* Surface Manager: Provides surface holders for various applications. [4]

Of the nine listed libraries, two were used extensively (OpenGL/ES and Surface Manager) and will be discussed in further detail in section 11.1.

### Emulator

Android applications can be run either on the physical device or on the Android Emulator, which is shipped with the Android SDK. The main benefits of utilizing the Android Emulator are twofold:

* Integrates very well with the Eclipse Integrated Development Environment (IDE) [5].
* Eliminates a fair amount of “real-world” testing, as the application programming interface (API) is nearly identical to a physical mobile device.

Of course, there are some functionalities which even the Android Emulator does not implement, such as the built-in gyroscope of more modern mobile devices. This particular issue is covered in section 11.3 (Development Issues and Solutions).

### Additional Libraries

In order to properly test the tilt functionality of the Android mobile device without access to such hardware, an open source project by OpenIntents was utilized. As mentioned in section 3.1.2, the Android Emulator does not implement or support the gyroscope functionality which the RoboWars project has taken advantage of. Fortunately, OpenIntents has already solves this problem by implementing their own open source project – Sensor Simulator – which directly addresses this issue.

The Sensor Simulator project allows the designer to “simulate sensor data with the mouse in real time. It currently supports accelerometer, compass, orientation, and temperature sensors, where the behaviour can be customized through various settings.” [6] As the API for this particular library is nearly identical to the API of the actual Sensor Event Listener class, modifying source code to use this library in place of its hardware counterpart is extremely easy and involves modifying only a couple of lines of code. [6]

## Lego Mindstorm NXT 2.0

### Robotic Requirements

The robots required for RoboWars were required to have four characteristics which were identified as crucial to the implementation of the project. The first characteristic is: zero radius turning, to enable the playing of the “light-cycles” game mode. The second characteristic required independent wheel/motor control to avoid having to build a complex steering system. The third requirement dealt with a wireless communication ability which provided the means to have controls being provided from either a server or mobile device, as well as allowing the robot to communicate back to the server. Finally the last requirement involved robot side position tracking. Robot side tracking was deemed the most crucial of the requirements as should a packet be dropped coming from the robot to the server the game model can still remain consistent. Had the server done the position tracking, and a packet had been dropped there could have been a model inconsistency which would render all usage pointless.

### Mindstorm NXT 2.0 Capabilities

The NXT 2.0 Mindstorm kits are the 3rd generation of LEGO Mindstorms. The kit consists of three main types of devices: servo motors, input peripherals, and the NXT Intelligent Brick. These devices, when working together alongside the LEGO bricks offers an almost limitless number of design possibilities. The NXT Intelligent Brick is a 48MHz microprocessor with 64KB of SRAM. It allows up to three servo motors and four input peripherals. The brick can be connected through a wired USB 2.0 connection or wirelessly over a Bluetooth 2.1. It also has an LCD screen and a speaker capable of 8Hz playback. The servo motors are motors controlled using on board tachometers. They provide up to 180RPM with 15N•cm torque [7]. The peripherals included in the kit are: two touch sensors, a colour sensor, and an ultrasonic sensor. The sensor used in our design is the LEGO colour sensor. This sensor consists of three devices: a white flood light, an infrared colour reader and a RGB LED bank for colour output and reading. This sensor can read values in full 8-bit RGB colour while countering for light saturation, and brightness to return the correct 0-255 RGB value as well, LEGO provides six pre-programmed colours: white, black, yellow, green, blue and red [8].

### RoboWars Robot Design

RoboWars has built its own custom robots for this project. In our design we have used two servo motors to provide both drive and steering, balanced over a central steering column. These two servo motors can provide zero radius turning by having them spin in opposite directions. To increase the maximum movement speed of the robots the design has added a gearing system with a 3.38:1 gear ratio. The robots communicate to the server over a custom built messaging protocol over a Bluetooth connection. The robots also have a front mounted colour sensor used to provide readings from the map to provide error corrections.

# Requirements

The RoboWars project was an original idea proposed by the project team, and as such no external client existed to participate in requirements elicitation. To ensure that the project scope and requirements were clearly defined requirement elicitation was performed within the project group, and the requirements were documented through the collection of use cases shown in Table 7.1 and Figure 7.1. A complete use case specification was produced, which has been included as Appendix A. This document provides a detailed description of the actors which interact with the system, as well as a formal specification for each use case. In addition, use case realizations (in the form of sequence diagrams) were also produced for selected use cases to aid in early design work. These sequence diagrams are included as Appendix B.

Figure .1: RoboWars use case diagram



Table .1: RoboWars use case names and descriptions

|  |  |
| --- | --- |
| **Server Administration Use Cases** | |
| **Name** | **Description** |
| **Configure wired camera** | The server administrator selects the camera to use from a list of detected cameras, and configures the position, facing, and field of view of the selected camera.  Example scenario: Alex opens the administrator GUI on the server machine, and selects a video feed to use from a list of detected cameras. Alex fills out fields specifying the position, heading, and field of view of the camera, and saves the settings. |
| **Display local video feed** | A live video feed of the “arena’ containing the remote robots is read from the attached camera and displayed through the server administration GUI.  Example scenario: Alex opens the administrator GUI on the server machine, and selects a “preview camera” option. A frame is displayed in the administrator GUI that shows the live video feed from the currently selected video source. |
| **Reset robot positions** | The server administrator forces a reset of the robot positions in the system’s internal state. This should be performed if the system’s positioning information for a robot becomes out of sync with the robot’s actual position for any reason.  Example scenario: Between games, Mike picks up one of the registered robots, and moves it to a new place in the robot arena. Mike opens the administrator GUI, and selects the moved robot. A prompt is displayed which allows Mike to enter a new position and heading for the robot. This data is saved, and used to update the virtual game state. |
| **Display simplified game state** | The server administrator triggers the display of a simple 2D rendering of the current game state which is visible in the server administration GUI.  Example: Mike and Steve have previously connected both Android clients and robots to the system, and have launched real time gameplay. Alex opens the administrator GUI, and selects an option to display a 2D graphical view. A new frame appears, which contains a simple “top down” rendering of the current game state (i.e. elements are drawn for the robots, and any virtual game elements such as walls or projectiles). |
| **Hardware Robot Use Cases** | |
| **Name** | **Description** |
| **Register robot for remote control** | A remote robot connects and registers with the system.  Example scenario: A Mindstorm NXT 2.0 robot running the RoboWars client software connects to the server through Bluetooth. The robot identifies itself as “NXT1” and is registered with the main server lobby. The identifier “NXT1” can now be seen in the robot list in the administrator GUI. |
| **Unregister robot for remote control** | A remote robot unregisters from remote control and disconnects from the system.  Example scenario: A Mindstorm NXT 2.0 robot previously registered with the RoboWars system sends a disconnect signal, and powers down. The server unregisters robot “NXT1”. The identifier “NXT1” can no longer be seen in the robot list in the administrator GUI. |
| **Update robot position** | A robot asynchronously updates the system to reflect its current position.  Example scenario: The robot with identifier “NXT1” previously registered with the RoboWars system. The robot receives a movement command, and rolls forward some distance. The robot calculates its new position, and reports the position back to the central server through Bluetooth. The new position and heading of the robot is used to update the server’s virtual game model. |
| **User State Management Use Cases** | |
| **Name** | **Description** |
| **Join waiting lobby** | A mobile client initializes a connection to the server.  Example scenario: Steve launches the RoboWars Android client software. Steve is prompted for an IP address, username, and port. The client uses the provided IP and port to connect to the RoboWars server and Steve’s username (“Steve”) is registered with the server lobby. “Steve” is added to the list of connected users visible in the administration GUI. |
| **Send chat message** | A mobile user sends a chat message to be displayed to all other connected mobile users.  Example scenario: Steve has previously connected to the RoboWars server. Steve types a chat message into the client side chat lobby, and presses the “Send” button. The chat message is sent to the server, and broadcast to all other players (prefixed by “Steve: “). |
| **Opt-out of robot control** | A mobile user who wishes to view rather than control the remote robots may choose to opt-out of robot control and become a pure spectator.  Example scenario: Steve has previously connected to the RoboWars server. Steve selects a “Spectator” option in the Android client GUI, and a message is sent to the server. The server updates Steve’s state to that of a pure spectator, and broadcasts a message to all connected clients indicating the change. |
| **Select game type** | A player selects the type of game to play prior to starting a new game.  Example scenario: Steve has previously connected to the RoboWars server. Steve selects a game type from a list of options in the Android client GUI. The selected game type is sent to the server, and if no other players submit a more recent request the next game launched will use the selected game mode. |
| **Pair player to robot for remote control** | A specific player is paired to a specific robot for the duration of a single game. Any commands from the player will control only the robot paired to the specific player.  Example scenario: Steve, Mike and Alex areall connected as Android clients, and the two robots “NXT1” and “NXT2” are registered with the server. When a game is launched Mike is paired to “NXT1”, and Steve is paired to “NXT2”. Since no robots remain, Alex is registered as a spectator. For the duration of the current game, all commands from Mike’s client are propagated to “NXT1”, and all commands from Steve’s client are propagated to “NXT2”. |
| **Virtual Game Play Use Cases** | |
| **Name** | **Description** |
| **Launch game** | A player in the lobby initiates a game. The lobby is replaced with an augmented video feed, and players are paired to robots for remote control. Virtual game play begins.  Example scenario: Steve and Mike are both connected as Android clients, and the two robots “NXT1” and “NXT2” are registered with the server. Steve selects the launch game option from the Android client GUI, and a launch game signal is sent to the server. The server pairs Steve and Mike to “NXT1” and “NXT2” respectively, and real time game play is launched. |
| **Display augmented video** | The live video feed of the camera overlooking the robot arena is combined with the internal game state representation to produce virtual elements overlaid on the live video feed.  Example scenario: Steve and Mike are both connected as Android clients, and the two robots “NXT1” and “NXT2” are registered with the server. Game play has previously been initiated by Steve. Steve’s Android client receives both a live video feed, and a continuous stream of the current game state. This data is used to render live video overlaid with 3D graphics on Steve’s Android client. |
| **Move robot** | A mobile player issues a command for a robot to move, which is passed to the robot by the system.  Example scenario: Steve and Mike are both connected as Android clients, and the two robots “NXT1” and “NXT2” are registered with the server. Game play has previously been initiated by Steve, who is paired to robot “NXT1”. Whenever Steve tilts his Android smartphone, the orientation of the phone is sent to the server, and used to send a corresponding movement command to robot “NXT1”. |
| **Fire virtual projectile** | A mobile player issues a command to fire a virtual projectile, which is created and simulated in the system.  Example scenario: Steve and Mike are both connected as Android clients, and the two robots “NXT1” and “NXT2” are registered with the server. Game play has previously been initiated by Steve. Steve taps an onscreen button on the Android client GUI, and a signal indicating a “fire” command is sent to the server. The server processes this command, and generates a new virtual projectile originating from robot “NXT1”. |
| **End game** | A game in progress is ended and all connected mobile users are returned to the game lobby. Any paired robots are freed and considered unpaired.  Example scenario: Steve and Mike are both connected as Android clients, and the two robots “NXT1” and “NXT2” are registered with the server. Game play has previously been initiated by Steve. Steve loses connection to the RoboWars server, and the server recognizes this disconnect as a game ending condition. Propagation of movement commands to the robots is stopped, and all players are unpaired and returned to the server lobby. |

# Overall System Architecture

The RoboWars system is designed as a centralized system, in which both robot and Android clients connect to a central server which manages the system state and propagates information between clients as required.

Figure .1: A deployment diagram of the complete RoboWars system. Note that multiple instances of the Robot and Android nodes can be serviced by a single Central Server node.



A deployment diagram demonstrating the distribution of subsystems among nodes is shown in Figure 8.1. The RoboWars system is distributed over three types of nodes:

**Central Server:** This node represents standard consumer desktop hardware, with the minor addition of a Bluetooth dongle to support communication with the Mindstorm NXT 2.0 robotics kits. A Model-View-Controller architecture is used to separate the main server application into three subsystems which are all deployed on this node. The controller subsystem is responsible for managing robot and Android client connections, broadcasting the real time video stream, managing the server lobby (which entails broadcasting chat messages, and launching / terminating real time game play), propagating robot position changes to the model, and continually triggering physics updates to provide real time physics in the virtual world. The model component is dynamically constructed whenever a new game is initialized, and is responsible for storing the current game state (including virtual simulated entities), broadcasting state changes to the other subsystems, and filtering all client commands to ensure that virtual game state will not be violated by physical robot movement. The view component observes both the model and controller components, and provides an admin interface through which the server and camera settings are configured, the chat lobby is monitored, and the current game and client connection state is displayed.

**Robot:** The robot node represents a Mindstorm NXT 2.0 Intelligent Brick. Up to two connected robot nodes have been tested, although the implementation is designed to support an arbitrary number of robots. The robot client is implemented as a single subsystem which is responsible for local position tracking and remote execution of commands from the central server. Although the RoboWars implementation is designed with a custom Mindstorm NXT 2.0 client in mind, any Bluetooth capable robotics kit could be adapted to this purpose.

**Android:** The Android node represents any Android 2.2 smartphone. Up to two connected Android nodes have been tested, although the implementation is designed to support an arbitrary number of Android clients (including spectators who do not directly control a robot). The Android client is implemented as two subsystems: the Android client, and the game model (which uses identical code as the server implementation for serialization compatibility). The Android client subsystem is responsible for transmitting chat and control messages to the server, receiving and rendering the real time video stream, rendering OpenGL graphics, passing orientation and touch screen input to the central server, and receiving and processing serialized instances of the game model from the central server. The model subsystem is continually updated to match the current server state, and is used to determine the positions of entities for use with OpenGL rendering.

# Server Implementation

The central server is the most complex node in the RoboWars system, and manages all communication flows between the Android clients and robots. See Appendix D for class diagrams of the server implementation.

## Controller – User and Robot State Management and Data Propagation

Figure .1: Class diagram of the main classes concerning user and robot state management in the server side controller component



The central component of the controller with respect to user and robot state management is an instance of the ServerLobby class. ServerLobby is a passive class that is not concerned with any real time aspects of gameplay, but rather manages lists of references to instances of UserProxy and RobotProxy (each of which represents a connected client), and controls the launching and termination of real time gameplay. The ServerLobby uses an event model to allow any class which implements the ServerLobbyListener interface to receive events whenever a player or robot joins or leaves the server, whenever a chat message is received, or whenever a game is launched or terminated. In the current implementation, this functionality is used primarily to allow the view subsystem to display lists of connected clients and the server chat lobby. The overall structure of these classes can be seen in Figure 9.1.

Figure .2: Sequence diagram of Android client registration



To manage incoming Android client connections, the controller uses an instance of TcpServer, an active class which runs a separate thread to continually accept incoming TCP connections on a specified IP address and port. Whenever a new connection is received, the reference to the client’s socket is used to construct a UserProxy instance. Each UserProxy is an active object which runs a dedicated thread to continually read new messages from the client’s socket. Once a UserProxy is generated, a handshake procedure is carried out to ensure the version of the RoboWars client is compatible with the server. The client must also provide a unique username, and the client will be required to select a new username if the username is already in use. Once the handshake procedure is complete, the user is registered with the server’s main lobby. Users can interact with the server lobby by sending chat messages, changing the game mode, changing their own ready or spectator status, and initiating game launches. In addition, once the handshake is complete all further communication is achieved by sending serialized event classes rather than raw strings, and the UserProxy registers as a listener on the ServerLobby. In this manner, instances of UserProxy listen on the main ServerLobby instance, and propagate events to their respective clients. A sequence diagram outlining this process can be seen in Figure 9.2. This could have been implemented without an event model (as the ServerLobby stores references to all connected instances of UserProxy and could call functions directly), but it was decided that an event model was preferable as an event model was already desired to avoid coupling with the view subsystem, and it is preferable if a standard communication path is used for both the view subsystem and instances of UserProxy.

Figure .3: Sequence diagram of robot client registration



Robot connections are handled similarly to Android clients, but rather than awaiting incoming connections, the server actively attempts to discover compatible Mindstorm NXT 2.0 clients within Bluetooth range. A sequence diagram outlining this process can be seen in Figure 9.3. This is accomplished using the PC libraries of LeJOS (“Java for LEGO Minstorms”) [9]. The functionalities of LeJOS in general will be discussed further in Section 7 (“Robot Client Implementation”). LeJOS provides a simple interface to enable either Bluetooth or USB communication with Mindstorm NXT 2.0s running LeJOS firmware. First, an NXTComm object must be generated by supplying a static method of the NXTCommFactory class with a constant specifying whether USB or Bluetooth communication is desired. In the case of Bluetooth communication, the NXTCommFactory will automatically select a compatible underlying implementation from one of several supported libraries based on the underlying operating system. RoboWars has been implemented and tested exclusively on Windows, and in this case LeJOS will select Bluecove [10] as its underlying Bluetooth implementation. Bluecove is an open source library implementation of the standard Java Bluetooth API specified in JSR-82. However, the details of the underlying implementation are largely unimportant for the purposes of RoboWars, as LeJOS provides methods to write and read from Bluetooth using the standard Java input/output stream API once a connection has been established. It is worth noting that the Bluetooth discovery process blocks for a significant duration (approximately 5 seconds for each detected Bluetooth device), and the BluetoothServer class is implemented such that robot redetection always runs in a separate thread from the caller. Once a connection is established, a RobotProxy instance is created and registered with the ServerLobby. RobotProxy is similar to UserProxy in that it is an active class that continually reads the Bluetooth input stream to receive position updates from its associated robot. When a game is in progress this position information will be propagated into the game model, and robot commands will be propagated through the RobotProxy to its associated robot.

Once both instances of UserProxy and RobotProxy are registered with ServerLobby, a game can be launched. Game launch requests must be issued by an Android client, and a UserProxy propagates the request to the ServerLobby. The ServerLobby will ensure that enough (non-spectator) players and robots are registered for the selected game type, and will generate a new GameController instance to represent the new real time game. The GameController is an active class which generates an encapsulated instance of the game model, and runs a thread to continuously update the model’s physics implementation. Users are selected for pairing with robots by the ServerLobby (based on which users have been paired most recently), and the ServerLobby generates an instance of ControlPair for each pairing. ControlPair class is a passive class used primarily for data encapsulation which stores a reference to a single UserProxy and RobotProxy. Instances of ControlPair are registered with the GameController, and once all control pairs have been registered with the GameController additional players are added as spectators. When this is complete, the game is launched. This generates a ServerLobbyEvent which signals to clients that a game has begun and OpenGL rendering should be displayed. A collaboration diagram outlining this process can be seen in Figure 9.4.

Figure .4: Collaboration diagram outlining the game launch process.



Once game play has been initialized, instances of UserProxy send tilt vectors and button input to the GameController where the data is used to generate instances of RobotCommand based on the control scheme specified by game mode in use. These commands are validated against the game model to ensure that game state will not be violated, and are then serialized and written to the user’s paired robot through the RobotProxy reference contained in the instance of ControlPair. Whenever the model’s game state changes, the GameModel generates GameEvent objects which are captured by the listening GameController, and passed to all connected instances of UserProxy.

## Controller – Real Time Video Streaming

### Alternative Solution – RTP/RTSP Streaming

Ideally, the real time video streaming in the RoboWars should use an established and well defined protocol. The first attempts to implement video streaming in RoboWars made use of the Real Time Transport Protocol (RTP) for data transmission, and Real Time Streaming Protocol (RTSP) for control signals. This functionality is well supported by the standard Android library, which provides a MediaPlayer class specifically designed to receive and display and RTP/RTSP stream [11]. To test this functionality, the Android client for RoboWars was modified to make use of the MediaPlayer class to accept and display an RTP/RTSP video stream.

The server side generation of the RTP data stream was partially implemented using the “Freedom for Media in Java” (FMJ) project [12], which provides an open source implementation of the standard Java Media Framework API [13] for capturing, playing, and streaming media. JMF provided the means to easily generate an RTP data stream, but further investigation determined that alternate means would need to be used to implement an RTSP server to generate the required control signals expected by the Android MediaPlayer. Rather than continue with server implementation, the open source VideoLAN (VLC) project [14] was used to generate RTP/RTSP streams purely to test the Android client implementation. A deployment diagram of this implementation can be seen in Figure 9.5.

Figure .5: Deployment diagram of the testing system used for RTP/RTSP streaming



Through this testing, it was determined that the capabilities of the Android MediaPlayer implementation were insufficient for the needs of the RoboWars project. Although the video stream could be reliably displayed, an unacceptable delay of approximately 10 seconds occurred between the transmission and rendering of the video stream. Further testing determined that the cause of this delay was the buffering behaviour of the MediaPlayer implementation. The MediaPlayer class is intended to be used for the streaming of relatively static media (ex. YouTube videos) rather than real time streams with low delay requirements. The implementation of MediaPlayer buffers video based on a fixed length of video, rather than data size, and does not provide any external API to modify this behaviour. Regardless of the resolution, encoding, or bit rate of the video stream generated by VLC the ten second delay on the client side remained constant. Further investigation of the Android source code revealed that this buffering functionality is implemented at the level of phone firmware. Modifications to the Android firmware would require that all users of RoboWars install custom firmware on their phones, and would limit RoboWars clients to phones that have had firmware specifically modified for RoboWars. This would have significantly reduced the generality and potential market for RoboWars, and as such further modifications were not attempted at the firmware level.

### Implemented Solution – Custom UDP Protocol

In order to provide real time video streaming with an acceptably low delay a custom protocol for data transmission was required. The use of the FMJ project to interface with the webcam was discarded (as the RTP stream generation was no longer required), and replaced with the simpler LTI-Civil library [15] (which is also used internally within FMJ). LTI-Civil is an open source library which provides a simple API for capturing images from video devices entirely independently of the JMF API.

Figure .6: Class diagram of classes responsible for media streaming



Video streaming is implemented through two classes in the controller subsystem: CameraController and MediaStreamer. A class diagram of the relevant classes can be seen in Figure 9.6. The CameraController class is responsible for storing the position, heading, and field of view of a specific camera, as well as opening and making available the instance of CaptureStream interface which is used to read images from the camera. The MediaStreamer class is responsible for discovering available cameras, maintaining a valid list of instances of CameraController, and maintaining a list of users that video packets should be served to. The MediaStreamer also provides methods to select an active camera from the list of CameraControllers, and methods to start and stop the active capture stream. The MediaServer is registered as a listener on the main ServerLobby instance, and automatically adds and removes users to its list of clients to serve whenever a user joins or leaves the main lobby. In addition, the MediaStreamer also receives game launch and termination events from the main ServerLobby, and the video stream is launched or terminating along with real time gameplay. Camera position information is attached to game launch events before they are transmitted to clients, ensuring that the position of the camera is known for OpenGL rendering purposes.

To serve video the MediaStreamer must receive a valid instance of CaptureStream from the active CameraController. At this point, the observer pattern is used to read frames from the webcam. MediaStreamer implements the CaptureObserver interface, which allows it to register as a listener on the active CaptureStream instance. Once the stream has been launched, the onNewImage() function of MediaStreamer will be called by LTI-Civil at a fixed rate (determined before the CaptureStream is launched).

Video is transmitted to clients as a series of individual frames encoded as JPEGs. The onNewImage() function of MediaStreamer receives frames from the webcam in an unspecified format internal to LTI-Civil which is converted into an instance of BufferedImage from the Java standard library using a static method of the AWTImageConverter class (provided by LTI-Civil). Once the frame is in BufferedImage format, a static method of the ImageIO class (from the Java standard library) is used to encode the frame as a JPEG and write the result to a byte array. This byte array is split into segments of a configurable size (set via a static constant in MediaStreamer), and each segment is packaged into a separate UDP DatagramPacket. Each packet carries three fields in addition to the video data payload:

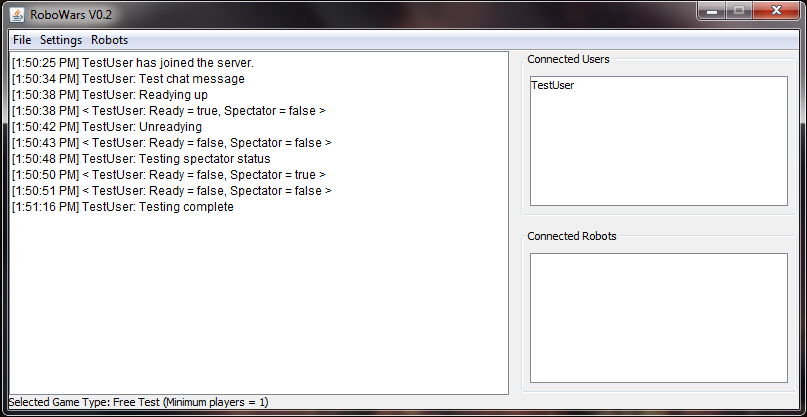
* **Frame number**: Specifies which frame this packet belongs to. In the current implementation this field alternates between 1 and 0, as it is unlikely that an individual segment will arrive more than a full frame out of sequence (and carrying the next segment number expected by the client). Even if this does occur the frame will be discarded as corrupt when decoding is attempted on the client side, and a single dropped frame is unlikely to be noticed by the user.
* **Segment number**: Specifies which segment of the current frame this packet carries. This field starts at 0 for the first segment of each new frame, and is incremented by 1 for each packet sent.
* **Last segment**: A Boolean field which is set to true only on the last packet of a given frame.

These generated packets are transmitted to all connected clients through UDP unicast on a specified port (the main server TCP port incremented by 1). UDP multicast would be preferable for this application as it reduces the overall bandwidth requirements (as packets are duplicated as required at the network level rather than at the server), and this method was originally implemented. However, the HTC Desire smartphones utilized by the project for testing were determined to be incapable of receiving multicast or broadcast packets, and as such the usage of UDP multicast could not be properly verified. This shortcoming is not officially documented, but is well corroborated by other developer reports [16]. Although UDP unicast is less bandwidth efficient, its usage allowed for the project to test video streaming functionality and perform demos with the available hardware.

The Android video client only supports the decoding of incoming video frames, and does not exchange any further information with the server (over UDP). A byte buffer is generated, and the video data from each incoming packet is appended to the buffer whenever a packet is received that matches the next expected frame and sequence number. When a packet with the “last segment” field is received, the buffer is decoded into the standard Android image format using the BitmapFactory class provided by the Android standard library. This bitmap is then passed to the OpenGL renderer to be loaded as a texture. The client’s buffer is cleared and the expected incoming frame and sequence numbers are reset whenever a new packet is received with a segment number of 0 (indicates the first segment of a new frame).

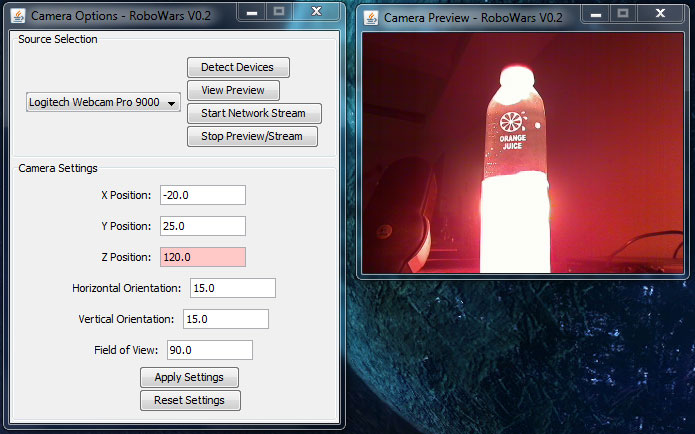
## View – Administration Panel and Camera Configuration

Figure .7: The administrator view of the RoboWars server lobby



The administration panel uses standard Swing libraries to provide a GUI for the administrator to configure the server. The main frame of the administration panel is implemented in the AdminView class, which subclasses JFrame. A screenshot of the GUI can be seen in Figure 9.7. The AdminView class implements ServerLobbyListener and listens for events from the main ServerLobby instance to update the connected user/robot lists and the main chat panel. The AdminView also contains a reference to the BluetoothServer instance, which is used to manually trigger robot redetection. This was originally implemented as a periodic function that did not require administrator interaction, but the period redetections were found to interfere with communication with robots during gameplay. The AdminView also generates an instance of CameraSelectionView which contains a reference to the MediaStreamer instance, and allows the administrator to configure the camera settings for the server. A screenshot of the camera configuration interface can be seen in Figure 9.8.

Figure .8: A screenshot of the camera configuration and preview GUI



## View – 2D Display

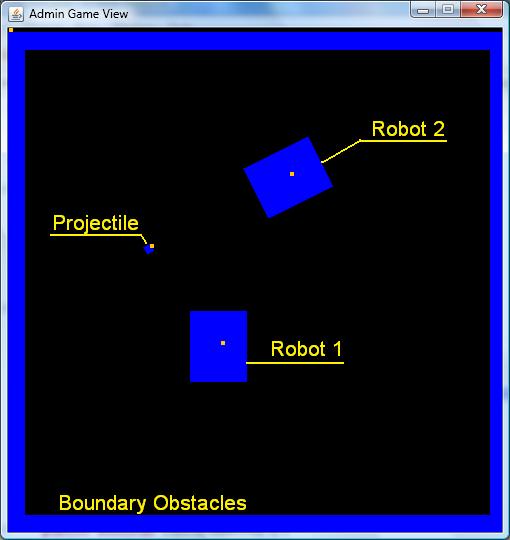


Figure .9: A screenshot of the 2D server game screen

The 2D display (contained within the class Admin2DDisplay) is meant to provide a simple 2D visual representation of the current state of the game and virtual world. It is a JFrame with just a Canvas as part of its content pane. GameEntities are drawn on the screen using the same coordinate system as the model; only it is always scaled to 500 pixels squared no matter the size of the actual game arena.

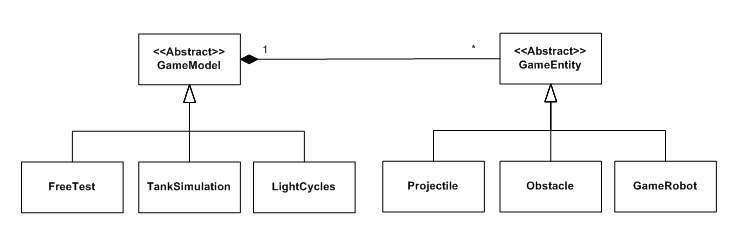
Java graphics allow for easy drawing of polygons with vertices as inputs, which works well with the model as GameEntities already represent their shape as an array or vertices.

The 2D display class implements GameListener, so every time the game state changes in any way, the canvas is redrawn and displayed, showing any change in position of any GameEntity.

## Model – Virtual World Model Implementation

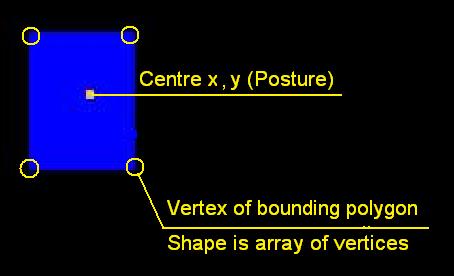
### Architecture

Figure .10: Top level architecture of the server model



The server model is the backend of the server. There are two classes that handle all basic functionality of the modeled virtual world. All data concerning the virtual representation of the robot and the associated virtual entities are stored within an abstract class called GameEntity. Its main purposes are to store the entity’s x and y location as well as its heading, as measured from the positive x-axis, exactly the same way position is stored on the robot client. It also stores the 2D polygonal shape of the entity, represented as an array of x and y coordinates relative to the central position of the entity. The shape of the entity not only serves as something to draw with on the 2D display, but more importantly it serves as the bounding area of the entity for which the collision detection system is concerned with. The large majority of its methods deal with collision detection, the mechanics of which is explained later in the section.

Figure .11: GameEntity visual representation



Position tracking and related functionality are common between different types of entities, but for specific behaviours and entity states, sub-classing is required. Currently, there are three types of GameEntities: GameRobot, Projectile, and Obstacle, each one with its own special fields and methods that dictate its behaviour in the virtual world.

All game logic and game state checking are performed by the abstract class GameModel. It is the top level class of the model and it is the only one the GameController has access to. The architecture is such that this abstract superclass contains all the base methods that its subclasses as well as the controller will need to access, while the subclasses that are actually instantiated represent the game type that was selected, and as such, implement the specific behaviour and fields required of that game type. There are currently three game play modes (and subclasses of GameModel by the same name): Tank Simulation, LightCycles, and FreeTest.

### Use of Java Event Model

It is necessary for the higher levels of the system to know when certain events happen within the model, for example when a robot fires a projectile, a collision is detected, or when the game is terminated. Thus, much like how the ServerLobby utilises event passing to manage robot or client connections and chat messages, the event model is used here to let the controller know when critical events happen in the currently running game.

Specifically, classes that need to listen to the GameModel implement an interface called GameListener. Two classes currently need to listen to the GameModel: GameController and the Admin2DGameView.

One final application of note is how the method of event passing is used with the Android client. Since the client is meant to have a local copy of the model content, the controller must have a way to pass the server model to the client every time the game state is changed. Using the event model, a reference to GameModel is always passed with every event that is generated. Thus, passing every event along the network provides the client with an updated GameModel every time it is changed.

### GameEntities and 2D Collision Detection

Every virtual world object, or GameEntity, has x, y and heading values to denote its position on the map. These values are conglomerated into a class called Posture. Additionally, an array of x and y values denote the vertices of the polygon shape relative to the map origin (they are supplied to the constructor relative to the Posture for easy visualisation of the shape, and then are compensated for the map origin). This polygon shape is used as the bounding area for collision detection.

Detection begins by finding the x and y differences between values of different vertices on both polygons, effectively giving the line segment vectors of the polygon’s “edges”. Then it enters a loop such that for each edge, both polygons are projected on the perpendicular axis of that edge. If at any point during this loop the projections on a given axis don’t overlap, then the two polygons do not collide. Only if the loop finishes and all projections overlap, is there a collision.

Compensating for rotation about the Posture is another challenge. Since the vertices are stored relative to the origin, temporarily relocating the Posture to the origin is needed. Then the x and y coordinates of the Posture are changed into polar coordinates r and θ to easily rotate the vertices about the origin. Each θ value is incremented by the amount needed, and then each vertex is changed back to rectangular coordinates and moved back to the original position of the Posture. Thus, the shape appears to have rotated by the degree value supplied.

### GameRobot and RobotCommand Priority

In order for a client controlled robot to physically react to something in the virtual world, client control must be overridden by something triggered in the model.

GameRobot has special functionality in the model. In addition to behaving as any other GameEntity, it also has the added function of storing a RobotCommand, which the GameController would access to see if it has any priority over RobotCommands issued by the client.

Sometimes the model generated RobotCommands override the ones from the client, such as when the robot collides with a virtual Obstacle. The GameModel would set the STOP RobotCommand in the GameRobot that collided with the wall. In this case, if the client wishes to move the robot forward, the model would not allow it because it is made so that the robot would not be allowed to pass through walls.

Other times it is the client generated RobotCommands that should override the ones form the model. An example of this is in the LightCycles game mode, where the default action of the robots would be the MOVE\_CONTINUOUS command. When no other command from the client is present, the robot would execute the model generated command. When the client sends a left or right turn signal, then in that case the client generated command would override the model command.

### Android Incompatibility with LeJOSPose

GameEntities used to use the LeJOS class Pose to store x and y position coordinates, and heading. It was believed that using the same position tracking data structure between the robot, server and client would reduce the need for adapters between modules. However, it was found that Pose makes a reference to the Java AWT class Point2D, and that Android does not in any way support or recognize the Java graphics library, which causes fatal exceptions to arise. This problem was detected very late in the final stages of module integration, so a quick, but feasible solution was needed.

The Pose class was already tightly integrated in all aspects of the system, but all that was needed was to get rid of the java.awt.Point2D reference. So class Pose was copied over into a new custom class called Posture, and the Point2D reference was replaced with a custom class called Vector. Other than a few minor changes to make Posture compatible with the system all that was needed was a few line changes to change the Pose objects incoming from the robot into new Posture objects for the server and Android client to understand.

## Server-Side Logging

The components of RoboWars deployed on the central server node make extensive use of the Apache Log4j open source logging library [17] in order to aid with debugging and performance testing. This carries a number of advantages over console logging, including:

* All logging statements are tagged with the thread identifier of the thread which initiating the logging, the class name of the class which initiated the logging, and the current system time (in milliseconds). This data is very useful to have in a standardized format when debugging and acquiring performance metrics.
* Up to 6 independent levels of logging are supported (TRACE, DEBUG, INFO, WARN, ERROR and FATAL), each of which can be configured separately.
* Logging configurations are modified through a configuration file which is read at runtime. Logs can be directed to console or file, and these settings can be set independently for each logging level (by default, RoboWars logs DEBUG messages to file, and INFO and ERROR messages to both console and file). Suppressing log output can be done through runtime configuration, rather than modification of source code.

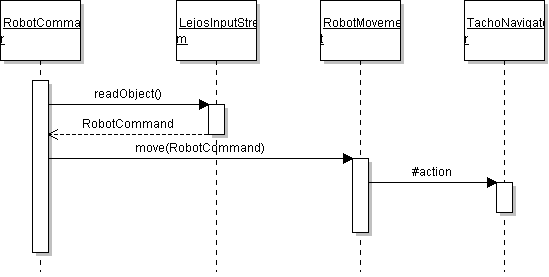
# Robot Client Implementation

## Client Implementation

The robot client is a rather simple structure. It consists of many classes running independently. They are as follows:

* RobotMain: The main class which is used to initialize the RobotCommandController and the listener for button press interrupts.
* RobotCommandController: Robot Command Controller acts as the main thread. It has the main representations of RobotMovement and ColorSensor, and further more it contains the thread which will periodically return the position of the robot. RobotCommandController’s main purpose is to receive incoming objects from the server over Bluetooth and call the corresponding functions in RobotMovement or ColorSensor. See Figure 10.1

Figure .1: How commands are received



* ColorSensor: The class used to generate and possess the RobotMap used to correct the errors through movement. In this class is where the error correction color reads are performed and processed.
* RobotMovement: This is the thread safe access point for the RobotCommandController to access the navigator classes (see Figure 10.2 for structure). It simply forwards the commands onto the navigator through calling the correct methods in a safe manner.

Figure .2: RoboWarsNavigator structure



* PositionTracker: This class simply returns to the server over Bluetooth the robot’s pose on a periodic interval through the LejosOutputStream and then returns to sleep.

## LeJOS and Modifications

### LeJOS

LeJOS is an open source firmware to provide a JVM which can be embedded on microcontrollers, thus allowing the microcontrollers to be programmed in Java [9]. Originally started in 1999 providing the firmware for devices which were sent to the international space station in 2001 LeJOS has continued to grow. Since 2006 LeJOS has be focused to the LEGO RCX and LEGO Mindstorm NXT bricks as its primary target microcontrollers. LeJOS features a large API as well as a vibrant community which is still active on a daily basis. Furthermore it features a large number of tutorials and reference materials to allow easy development. Although this seems ideal there were many occasions where there were inconsistencies which required major modification, which will be expanded upon in the following sections

### Bluetooth Communications and I/O Modifications

LeJOS provides by default an effective handshake system as well as a means with which to provide input and output through subclasses of the Java input and output streams. LeJOS also boasts the ability to provide Object passing over these input and output streams. Unfortunately these boasts are unfounded; although a potential system was started it was never fully implemented. Thus a new system was required to be implemented to allow object passing. From this new implementation our own LejosInputStream and LejosOutputStream were born as well a new message protocol scheme was implemented.

#### LejosInputStream

LejosInputStream is used to receive and decode packets sent over the server to the robot. It receives a byte stream over Bluetooth (or any other InputStream subclass), which it records into a byte array. This array is then converted into a plain text string. The plain text string is then decoded and new objects are created and returned. This input stream though is unable to decode all objects, only those which are used in RoboWars (RobotCommand, RobotMap, Vector<Int>, and Pose).

#### LejosOutputStream

LejosOutputStream is used to decode and transfer objects to and from the robot. All objects which can be sent and decoded successfully over the two streams have a function which returns a designated output string in the correct message format. This string is then converted into byte representations of the text. From there the newly formed Byte[] is transferred a byte at a time using the standard OutputStream.write(Byte b) function in java.

#### RoboWars Message Protocol

In RoboWars we have implemented a message protocol to allow the transfer of particular objects over LejosInput/OutputStream. At the moment four objects can be sent over the protocol: RobotCommand, RobotMap, Vector<Int>, and Pose. They are turned into transferable objects by calling the corresponding toOutputStream function. The message protocol for each object is as follows:

* RobotCommand: [<type>|<speed>|<turnbearing>|<special flags>|<pose>]
* RobotMap: [[MapPoint0],[<x>|<y>|<color>],….,[MapPointN]]
* Vector<int>:[<r-value>|<g-value>|<b-value>|<saturation>]
* Pose: [<x-cooridinate>|<y-coordinate>|<heading>]

### Navigation and Piloting Modification

LeJOS by default provides a system to track position and heading of the robot in multiple ways. The main two tools used are the HiTechnic sensor compass paired with the LeJOS CompassNavigator class or using the onboard tachometers in each servo motor coupled with the LeJOS TachoNavigator class. Due to the hardware available to the project we were required to use the tachometers embedded in the servo motors for our positioning. These navigators were used to provide high level position tracking as well as act as the access point to the Pilot class which actually operates the motors. As the project progressed there became a series of issues and challenges that needed to be overcome and corrected. They are identified and explained in this section.

#### Blocking on a steer() Call

In order to implement the rolling turns commands that are provided to the robot from the server as a representation of the movements done by the Android smartphone we were required to use the steer function provided by LeJOS. The steer function was designed to have the servo motors rotate with varying degrees of power dependant on a provided ratio. This ratio was from [-200,200] with 0 being drive directly forward, and ±200 being a zero radius turn in either direction. There was a crucial error though in the TachoPilot implementation of steer. The Pilot and all threads would block whenever a non-zero value was provided to steer. This was caused by the default TachoPilot doing comparisons between the provided float values and hard coded integer values on its branching decisions. This issue resulted in the code driving itself into an infinite loop as it tried to calculate the actual turn ratios. This issue was overcome by modifying the hard coded values to reflect the true types of inputs provided. Thus the RoboWarsTachoPilot was born. By correcting this issue the robot was now free to operate in an 180o field as required for the LightCycles game mode.

#### 360 Degree Controls and Tracking

One of the two game modes that are implemented for the project is TankSimulation. TankSimulation is a simulation of the robots acting as tanks thus the robots needed to be able to go forwards backwards and any way in between. Unfortunately the code provided in the PilotClass used to drive the motors was only capable to drive the robot in a 1800 range thus not allowing the ability to travel backwards. To overcome this issue a new steer function was implemented which was able to account for the throttle provided by the user. Should the throttle be a value less than zero the parity bits for both servo motors were flipped causing them to rotate in the opposite direction. This allowed for 360o motion to be performed by the robot. There was an unintended consequence of this change though. The position tracking system was not designed to operate in a 3600 environment, and considers everything forwards. Thus resulting in a complete failure of the tracking, caused by two main components:

* The robot no longer knew what way was up: As it had no frame of reference for which way was forwards (usually provided by the parity bit) the robot would always assume that it was travelling forwards. To overcome this difficulty the UpdatePosition function in SimpleNavigator had to be rewritten to account for the parity of the motors in reference to their original state.
* On a context switch from forwards to backwards the position information would become invalid: This issue was caused by other functions that would call UpdatePosition before the change in parity could be taken into account. The robot would believe it had turned very far angles (>4000) even though the robot would only track from ±1800, furthermore the robot would lose track of where it was causing the (X,Y) co-ordinate to be meaningless. To overcome this whenever a context switch occurs the pilot is reset and then told where it is. Thus it believes that it started at that point and has travelled no distance at all.

## Position Tracking and Error Correction

### Position Tracking

RoboWars robots use dead reckoning to determine their location. Dead reckoning uses a starting location and a displacement for the robots to be able to figure out where they are. This is performed by the Navigator classes provided by LeJOS. The Navigator is used to update the Pose representation of the robot (x co-ordinate, y co-ordinate, and heading). The default Navigator does not take into account for the geared wheels so the displacement functions in SimpleNavigator had to be rewritten to overcome this difference. The displacements are based off of degree turns of the internal tachometers for each servo motor, and are then converted into units provided by the user. In our project the units are all based in cm.

### Error Correction and the Grid

As the robots move about the world, errors are introduced into the position tracking. Although we have tried to make this as small as possible it is impossible to remove all errors. At the moment there is a drift of ±2 mm in either direction as well as a shift of ±2o on the heading per 30 cm travelled. Although this is small it is still an issue that needs to be overcome. Despite our grandest efforts there is no way to conclusively correct the heading error if our tracking system is based off of tachometer rotation due to wheel slippage. To correct these errors, a complex system was created to track the robots with in a world. This system is known as the grid. The grid is a grid of dots with 1.5 cm diameter in varying colors placed on the game mat. This is a real world representation of an onboard RobotMap that the robot generates based upon inputs provided by the server or as the default setting. When a robot using its colour sensor drives over a dot and detects its colour the robot will determine where the nearest dot of that colour in the correct direction is and adjust its position accordingly. This “snapping” action snaps to the location of the center of the robot not the sensor, so this generated value is generated by the Robot using basic trigonometry.

## Future Suggestions

Due to technology available at the time, we have not been able to create a perfect tracking system. Yet alternatives to our design have been devised should there be a chance to improve upon it. These suggestions are as follows:

* Heading Correction: At the time we started there was not the budget to get the HiTechnic Compass sensor. In any future designs the robots should be equipped with such a sensor the heading error is removed. This will help in both the position tracking but also the snapping action done by the colour sensors as they use the heading during trigonometry.
* Digital Correction: When we began certain technologies were not released yet. The main important one would be the Microsoft Kinect. The Kinect projects an RGB grid. If we were able to project a similar grid on the game mat we would be able to get perfect position tracking using similar methods as are currently implemented [18].
* Gradient Colour Scheme: Due to the means available to us we are unable to implement this idea in its fullest. By making the grid into a colour gradient we would be able to get the correct position based off the RGB value read by the sensor. Because of the sensitivity we would be able to get an absolute position but will still be unable to correct the heading error.

# Android Client Implementation

## Android Libraries and Application Layout

### Android Libraries

As mentioned previously, the two main libraries utilized in this project (on the client side) are the OpenGL/ES library and the Surface Manager library.

#### Surface Manager

Android’s SurfaceManager is quite elegant in that it does not draw directly to the screen buffer, but rather forms the entire screen layout before drawing anything at all. In doing so, the developer has much more flexibility in terms of implementing graphical/visual effects such as transparent windows and transitions.

There are many built-in view layouts, such as a RelativeLayout, LinearLayout, and TableLayout, all of which were utilized in the RoboWars project. All of the layouts support all of the built-in widgets, and are very easy to understand, implement and use.

#### OpenGL/ES

The OpenGL/ES library shipped with Android is extremely useful, versatile and relatively well-documented. Although support specifically for OpenGL/ES is slightly more difficult to find than other Android documentation, OpenGL/ES conforms to the OpenGL 1.5 standard, which is very well documented and widely used. Section 11.2 discusses further detail about OpenGL/ES.

### Application Layout

The layout of the RoboWars Android application is relatively simple. The majority of the view components and characteristics are stored in an XML file; this approach keeps the source code clean and simple, and at the same time improves reusability. An example of how elegantly XML is incorporated into Android is provided at the end of this section (Figures 11.1 and 11.2). The remainder of the classes are generally models, for storing data, and controllers, for handling user input. A class diagram of the layout of the Android client is found in Appendix D. Essentially, the RoboWars class acts as the application controller; it holds references to the model and the TcpClient thread, and determines which courses of action to take when any event (either touch event or network packet event) occurs.

Figure .1:A simple LinearLayout implementation using only XML

\*Example adapted from the LinearLayout tutorial from the Android Developers website: <http://developer.android.com/resources/tutorials/views/hello-linearlayout.html>



Figure .2: XML code used to obtain the GUI as seen in figure 11.1 above

<?xml version=”1.0” encoding=”utf-8”?>  
<LinearLayout xmlns:android=”http://schemas.android.com/apk/res/android”  
    android:orientation=”vertical”  
    android:layout\_width=”fill\_parent”  
    android:layout\_height=”fill\_parent”>  
  
  <LinearLayout  
      android:orientation=”horizontal”  
      android:layout\_width=”fill\_parent”  
      android:layout\_height=”fill\_parent”  
      android:layout\_weight=”1”>  
      <TextView  
          android:text=”red”  
          android:gravity=”center\_horizontal”  
          android:background=”#aa0000”  
          android:layout\_width=”wrap\_content”  
          android:layout\_height=”fill\_parent”  
          android:layout\_weight=”1”/>  
      <TextView  
          android:text=”green”  
          android:gravity=”center\_horizontal”  
          android:background=”#00aa00”  
          android:layout\_width=”wrap\_content”  
          android:layout\_height=”fill\_parent”  
          android:layout\_weight=”1”/>  
      <TextView  
          android:text=”blue”  
          android:gravity=”center\_horizontal”  
          android:background=”#0000aa”  
          android:layout\_width=”wrap\_content”  
          android:layout\_height=”fill\_parent”  
          android:layout\_weight=”1”/>  
      <TextView  
          android:text=”yellow”  
          android:gravity=”center\_horizontal”  
          android:background=”#aaaa00”  
          android:layout\_width=”wrap\_content”  
          android:layout\_height=”fill\_parent”  
          android:layout\_weight=”1”/>  
  </LinearLayout>

</LinearLayout>

## OpenGL Rendering

OpenGL/ES is a cross-platform API that renders both 2D and 3D graphics on embedded systems. Theoretically, any OpenGL program created under the OpenGL 1.5 standard (or earlier) should work in OpenGL/ES, therefore there exists plenty of documentation for OpenGL/ES that is relevant to mobile developers [19].

Rendering 3D objects in OpenGL is relatively straight-forward, yet takes some time to learn and adjust to. An OpenGL drawing starts as a set of vertices, which define the corners of the polygon. Next, a set of index triplets are supplied; these define the order in which to connect the vertices. When texturing is enabled, another set of indices must be supplied, these ones relative to the polygon and not an absolute coordinate system. Once the shape of the polygon is defined, it must be placed and rotated. It is easier to think of the OpenGL surface as moving and not the actual polygons; placing a polygon at (0,0,0) will place it exactly where the previous polygon was placed. Further, once one polygon has been rotated and placed, all subsequent polygons will retain the same rotation. This is due to the fact that the polygons are not being rotated at all, but the OpenGL surface is.

## Development Issues and Solutions

### Android Emulator

Two prominent issues with the Android Emulator came up, both during implementation and during testing. The first, and most easily handled, was the fact that the Android Emulator has no built-in gyroscope emulator or stub. As the RoboWars mobile application depends on the tilt controls for steering during a session, either the application needed to be tested on a physical mobile phone, or an alternative software solution was needed. After some research an open source library named SensorSimulator [6] was found and used. Not only was this library very well documented, it also turned out to be very easy to implement into any existing project. Assuming source code already exists in the project which uses the hardware gyroscope, only a couple of lines of code require modification in order to switch between the SensorSimulator and the actual hardware gyroscope. The application allows the user to control the tilt of the phone along all three axes through a software interface. The application communicates with the RoboWars project through a TCP connection, so information transfer was not an issue, as it was already fully implemented and working.

The second, and more prominent issue regarding the Android Emulator, is the way it handles the OpenGL interface. Unfortunately, the emulator only draws the very first frame of an OpenGL scene, which means that none of the rendering or moving of objects can be tested over the Android Emulator. Unfortunately a solution was never found for this problem; the remainder of OpenGL testing was done through actual Android hardware as opposed to the emulator.

# Testing

## Robot Client Testing

To aid in testing the project is extremely modular. Continuing with the modularity the client can be identified into two sections, the I/O streams that are connecting the robot to the server and the code to run the robots. Both systems have been given the means to be tested as follows:

* **IOTest.java:** This script runs a simulation to check that the LejosInput/OutputStreams are able to successfully encode and decode every type of object that is used in communication for RoboWars matches.
* **TestColourSensor.java:** This script is used to check if the Colour sensor is still working properly. ColorSensor in RoboWars can be run in test mode where it returns the read and these reads are output on a server screen. Unfortunately this output must be approved by a human oracle.
* **KeyController.java:** This script is used to check that the robot is moving correctly. It acts as an interface to run all commands used by RoboWars over Bluetooth. The movements must be verified by a human oracle as there is no way to assert that the correct movement occurred programmatically.

## Android Client Testing

The Android client was tested mainly through the use of the Android Emulator software, and later via physical mobile devices. During the first half of the project, in testing various elements of the design of the Android application such as scrollbar functionality, button presses, and basic user interaction, it was crucial to take advantage of the Android Emulator; this allowed for quick, easy and efficient testing of the application’s interface without any prior knowledge or experience in Android application development.

## Server Side Testing

There are two types of testing as far as the server is concerned. One involves testing the internal mechanics and ensuring no bugs are present there, the other type relies on integrating the disparate modules of the system together and ensuring that all signals passed between them are compatible with each other.

### Controller Testing

Unit testing was performed on the controller component of the server using the open source JUnit 4 testing framework [20]. Test coverage was primarily directed toward the control classes of the component, while boundary classes such as UserProxy and RobotProxy were primarily tested through interaction with the actual boundary actors or simulators thereof. Classes used only for data encapsulation (such as ServerLobbyEvent and its subclasses) were also not specifically selected for unit testing, as they contain no significant implementation other than getters and setters. The classes in the controller component specifically selected for unit testing are: CameraController, ClientCommand, GameController, MediaStreamer, and ServerLobby.

To aid with unit testing, an automated code coverage tool called EclEmma [21] was utilized. EclEmma operates as a plug-in for the Eclipse IDE [5], and generates reports of JUnit test case coverage using “lines of code” coverage as the primary metric. In addition to generating coverage reports, EclEmma also provides real-time source code highlighting to better display coverage information. This functionality is particularly useful, as it provides immediate and detailed feedback to the developer whenever tests are run. An example of the output produced by EclEmma can be seen in Figure 12.1.

Figure .1 – An example of coverage output from EclEmma



EclEmma reports overall “lines of code” coverage of the controller module at 75.5% of lines covered. However, this value includes coverage of the test classes themselves, as well as coverage of classes which were not selected for unit testing. When considering only the five classes specifically selected for unit testing, test coverage increases to 83.2% of lines covered. It should be noted that the tests of MediaStreamer do not attempt to test the actual transmission of video data, which must be manually verified by a human oracle by viewing the transmitted stream on the Android client side. However, all other features of the MediaStreamer are tested (starting and stopping the video stream, registering and unregistering clients, reading frames from the webcam, etc.).

A software simulation of the Mindstorm NXT 2.0 robots was created to aid with unit testing functionality which requires a connected robot. The simulated robots are implemented in the TestRobotProxy class, and run identical navigation code as the actual hardware robots. The only difference is that rather than the navigation code interacting with an actual tachometer in the robot’s servos, a simulated software tachometer is used (implemented in TestTachoMotor). This allows unit tests to test functionality requiring registered robots, and also allows the system as a whole to be manually tested even when robots are not physically available. Software robots also do not have position tracking difficulties related to physical phenomena (such as wheel slippage), and therefore are ideal for testing that position tracking algorithms are functional.

Boundary classes were tested through separate means from the control classes. RobotProxy was primarily tested through manual verification of communication with the actual Mindstorm NXT 2.0 hardware. To test UserProxy, a separate TCP client simulator was created to simulate connections from Android clients. The simulator (implemented in the ClientSimulator class) features an interactive text terminal, and uses the parse function of ClientCommand to convert text strings into the serialized format expected by instances of UserProxy. This allows any command expected by the server (including gameplay commands with associated orientation vectors) to be generated locally without any access to the Android client. The client simulator also logs all communications with the server (both incoming and outgoing) to a text file, providing another source of verification for the server communication protocol.

### Model Testing

In the case of the collision detection model, in order to have exhaustive testing of the algorithm one would need to have as test input every possible combination of entity shape, number and order of vertices, permutated by all possible positions and rotations of the two entities in question.

A much more feasible, if not as precise way of testing is to instantiate and run the model independently of the controller (create a stub of the controller to simulate the signals the model needs) and to manually control the virtual entities at the keyboard to test the basic cases of collision detection.

The reasoning behind this method of testing is that the collision detection does not have to be perfect. Spending the time to design an extremely precise test framework is too expensive because all that is required is that collision to be detected when two polygons look somewhat too close together. If it is plausible that two polygons should collide to the naked eye, and it is indeed colliding according to the test, then the test is a success.

### Integration Testing

Integration testing is putting the whole system together for full system testing and working out the faults that arise between modules. At this point, all internal functionalities are working correctly.

This type of testing was found to be most useful with revealing flaws not with the system itself, but rather the problems to do with the referenced libraries. For example, only when the whole system was put together did the problem with Android not accepting java.awt arise, as well as the problem with the steer function in LeJOS.

# Conclusion

Through rigorous work we have completed the project based upon the original specs and recommendations. We believe that we have successfully implemented a system that provides an intuitive means to control robots using a general purpose mobile device. We have been able to achieve these goals through implementing centralized server software which manages connections, propagates commands and messages, and records client state from Android clients (via standard IP) and Mindstorm NXT 2.0 robotics kits (via Bluetooth). Furthermore Mindstorm NXT 2.0 robots were configured to support duplex Bluetooth communication, remote execution of commands received from the central server, dead reckoning position tracking and error correction, and real time position and heading data transmission to the central server. A Server side virtual world implementation which is capable of asynchronously updating robot positions, filtering client commands based on validity against the current game state, and generating outgoing robot commands to ensure physical robot movement does not violate the virtual game state. To enable long distance robotic control real time video streaming from a USB webcam connected to the central server to Android clients using a custom UDP protocol was successfully implemented. Finally Android client software which supports OpenGL rendering of an incoming real time video stream, as well as rudimentary three dimensional graphics rendering (simple textured polygons) was created to support our mobile goals. While implementing these features we were able to overcome many challenges. These challenges were made easier through our rigorous design phase as well as the modularity of our system. Furthermore we were able to validate our results through rigorous testing criteria. Through completing this project we have been able to evaluate situations where the current system could be improved but due to time, budget or scope constraints we were unable to implement these lofty goals. These include the recommendations for improvements in the position tracking of the robots, improvements to the Android Client Interface, and streamlining the game models. Should these steps be implemented we feel there would be a much more effective system.

# References

[1] “Git – Fast Version Control,” http://git-scm.com/ Accessed Mar. 15, 2011.

[2] “Github,” https://github.com/ Accessed Mar.15, 2011.

[3] *An Introduction to Android,* Huang Huguang, Inha University. Accessed Nov. 2, 2009.

[4] “What is Android?”, http://developer.android.com/guide/basics/what-is-android.html Accessed Mar. 23, 2011.

[5] “Eclipse – The Eclipse Foundation open source community website,” http://www.eclipse.org/ Accessed Mar. 31, 2011.

[6] “Sensor Simulator for simulating sensor data in real time”, http://code.google.com/p/openintents/wiki/SensorSimulator Accessed Mar. 23, 2011.

[7] “NXT Motor Internals”, http://philohome.com/nxtmotor/nxtmotor.htm. Accessed March 15, 2011

[8] “Class ColorLightSensor”, http://lejos.sourceforge.net/nxt/nxj/api/lejos/nxt/ColorLightSensor.html. Accessed Mar. 15, 2011

[9] “LeJOS, Java for LEGO Mindstorms”, http://lejos.sourceforge.net/ Accessed Mar. 15, 2011.

[10] “Bluecove JSR-82 Project”, http://bluecove.org/ Accessed Mar. 15, 2011.

[11] “MediaPlayer | Android Developers”, http://developer.android.com/reference/android/media/MediaPlayer.html Accessed Mar. 15, 2011.

[12] “FMJ”, http://fmj-sf.net/ Accessed Mar. 15, 2011.

[13] “Java SE Desktop Technologies – Java Media Framework API (JMF)”, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140239.html Accessed Mar. 15, 2011.

[14] “VideoLAN”, http://www.videolan.org/vlc/ Accessed Mar.15, 2011.

[15] “LTI-Civil”, http://lti-civil.org/ Accessed Mar. 15, 2011.

[16] “Multicast on Android 2.2”, http://stackoverflow.com/questions/3623143/multicast-on-android-2-2 Accessed Mar. 15, 2011.

[17] “Apache log4j 1.2”, http://logging.apache.org/log4j/1.2/ Accessed Mar. 15, 2011.

[18] “Microsoft Connect Review”, http://www.techradar.com/reviews/gaming/games-consoles/controllers/microsoft-kinect-905010/review. Accessed March 15, 2011

[19] “OpenGL ES”, http://www.khronos.org/opengles/ Accessed Mar. 23, 2011.

[20] “Welcome to JUnit.org! | JUnit.org,” http://www.junit.org/ Accessed Mar. 31, 2011.

[21] “EclEmma – Java Code Coverage for Eclipse,” http://www.eclemma.org/ Accessed Mar. 31, 2011.

# Appendix A – Schedule Gantt Charts

## A.1 – Original Schedule (from Project Proposal)



## A.2 – Revised Schedule (Revised for Progress Report, Dec 6th, 2010)



# 

# Appendix B – Use Cases

## B.1 – Actor Definitions

**Server Administrator:** The server administrator actor represents a user who is physically present at the location of the robot arena and server. This user has access to a configuration GUI running on the server machine, and is primarily responsible for configuring the camera to be used with the system and ensuring that the robots are correctly positioned prior to mobile users initiating gameplay. The server administrator can also handle any physical problems with the robots, and may force a reset of robot positions if necessary.

**Robot:** The robot actor represents any robot which will be registered for remote control with the system. Although the system is specifically designed for use with LEGO Mindstorm NXT 2.0 robotics kits, any robot which supports Bluetooth communication and the system’s custom communication protocol can be used.

**Mobile User:** The mobile user actor represents any user of the system who connects remotely from a mobile platform. Note that the module of code running on the mobile user’s smartphone is considered within the scope of the robotics control system for the purposes of these design documents. All mobile users have the ability to connect to the system lobby and chat with other users.

**Player:** A player is the default state of all mobile users when they connect to the system. This actor represents a mobile user who intends to make full use of the robotics control system. They are included in the rotation of users who are paired to the registered robots, and they have the ability to modify game options and launch games.

**Spectator:** A spectator represents a mobile user who has specifically opted-out of robotics control. Spectators can still view gameplay in progress and chat with other users, but they will not be considered in the robot control rotation.

## B.2 – Use Case Specification

**Server Administration Use Cases**

|  |  |
| --- | --- |
| **Use Case Name** | **Configure wired camera** |
| **Description** | The server administrator selects the camera to use from a list of detected cameras, and configures the position, facing, and field of view of the selected camera. |
| **Precondition** | The server software is running. |
| **Primary Actor** | Server Administrator |
| **Secondary Actors** | None |
| **Dependencies** | INCLUDE: **Display local video feed** |
| **Basic Flow** | 1. The server administrator selects a “configure camera” option from the server administration GUI. 2. The server administrator selects the desired camera from a list of cameras detected by the system. 3. The server administrator sets the co-ordinates and heading of the camera (in relation to the robot’s arena). 4. The server administrator specifies the field of view of the selected camera (necessary for augmented video rendering calculations). 5. The system displays the camera’s video feed through the server administration GUI (INCLUDE USE CASE **Display local video feed**). |
| **Post Condition** | A valid camera is configured for use with all further activities requiring a video feed. |

|  |  |
| --- | --- |
| **Use Case Name** | **Display local video feed** |
| **Description** | A live video feed of the “arena’ containing the remote robots is read from the attached camera and displayed through the server administration GUI. |
| **Precondition** | The server software is running.  A valid camera must have been previously configured for use with the system. |
| **Primary Actor** | Server Administrator |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. The server administrator selects a “view video feed” option from the system’s server administration GUI. 2. A video stream is read from the system’s wired camera. 3. The video stream is displayed in a separate frame of the system’s administration GUI. |
| **Post Condition** | The video stream is visible through the server administration GUI. |

|  |  |
| --- | --- |
| **Use Case Name** | **Reset robot positions** |
| **Description** | The server administrator forces a reset of the robot positions in the system’s internal state. This should be performed if the system’s positioning information for a robot becomes out of sync with the robot’s actual position for any reason. |
| **Precondition** | The server software is running.  A game is not in progress. |
| **Primary Actor** | Server Administrator |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. The server administrator selects a “reset robot positions” option from the server administration GUI. 2. The position of all robots currently registered with the system is reset to their default starting positions (this should be accompanied by the administrator physically relocating the robots to their starting positions). |
| **Post Condition** | The system records all registered robots as being positioned at their default starting position. |

|  |  |
| --- | --- |
| **Use Case Name** | **Display simplified game state** |
| **Description** | The server administrator triggers the display of a simple 2D rendering of the current game state which is visible in the server administration GUI. |
| **Precondition** | The server software is running.  A game isin progress. |
| **Primary Actor** | Server Administrator |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. The server administrator selects a “show game state” option from the server administration GUI. 2. The system displays a “game state” rendering window. 3. The “game state” rendering window is continually updated by the system to show a simple, overhead, 2D rendering of all virtual game elements currently in play. |
| **Post Condition** | The game state of the game currently in progress is visible in the system’s server administration GUI. |

**Robot Interaction Use Cases**

|  |  |
| --- | --- |
| **Use Case Name** | **Register robot for remote control** |
| **Description** | A remote robot connects and registers with the system. |
| **Precondition** | The server software is running. |
| **Primary Actor** | Robot |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. The robot initiates a Bluetooth connection with the server. 2. The robot sends a string indicating which version of the command and positioning protocol it supports. 3. The system responds with an acknowledgement. 4. The robot is added to the system’s list of active remote robots. |
| **Post Condition** | The robot will be considering for pairing to mobile users after registration has completed. |

|  |  |
| --- | --- |
| **Use Case Name** | **Unregister robot for remote control** |
| **Description** | A remote robot unregisters from remote control and disconnects from the system. |
| **Precondition** | The robot has been previously registered for remote control. |
| **Primary Actor** | Robot |
| **Secondary Actors** | None |
| **Dependencies** | INCLUDE: **End game** |
| **Basic Flow** | 1. The robot sends an “END” signal to the system. 2. The system responds with an acknowledgment. 3. The Bluetooth connection is torn down. 4. Any game in progress using the robot is ended (INCLUDE USE CASE **End game**). |
| **Post Condition** | The system does not consider the robot for any further pairing with remote users. The remote may be disabled once it has unregistered from the system. |

|  |  |
| --- | --- |
| **Use Case Name** | **Update robot position** |
| **Description** | A robot asynchronously updates the system to reflect its current position. |
| **Precondition** | The robot has been previously registered for remote control. A game is in progress. |
| **Primary Actor** | Robot |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. Any movement of the robot’s motors triggers a position update. 2. The robot calculates its current position based on a known last position and calculated change in position (depends on movement mechanism). 3. The robot’s current position is transmitted to the system through the Bluetooth connection. 4. The system’s internal state is updated to match the robot’s reported position. |
| **Post Condition** | The robot position stored by the system is synchronized with the position locally calculated by the robot. |

**Server State and User Management Use Cases**

|  |  |
| --- | --- |
| **Use Case Name** | **Join waiting lobby** |
| **Description** | A mobile client initializes a connection to the server. |
| **Precondition** | The server software is running. |
| **Primary Actor** | Mobile user |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. The mobile user initiates a connection to the system through standard internet protocol. 2. The system responds with an acknowledgment to ensure the connection has been established. 3. The mobile user provides a user name to identify them in the current session. 4. The mobile user is added to end of the rotating list of active users. 5. The mobile user is presented with the server lobby (a list of connected user names and a chat window). |
| **Post Condition** | The mobile user is identified by a unique user name and is added to the server lobby. |

|  |  |
| --- | --- |
| **Use Case Name** | **Send chat message** |
| **Description** | A mobile user sends a chat message to be displayed to all other connected mobile users. |
| **Precondition** | The mobile user has established a connection to the server.  The mobile user is either in the game lobby or in a running game. |
| **Primary Actor** | Mobile user |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. The user enters a string of text to be displayed to other mobile users. 2. The system displays the text message to all connected users along with the user name of the originating mobile user. |
| **Post Condition** | The chat message is displayed on the mobile client of all connected mobile users. |

|  |  |
| --- | --- |
| **Use Case Name** | **Opt-out of robot control** |
| **Description** | A mobile user who wishes to view rather than control the remote robots may choose to opt-out of robot control and become a pure spectator. |
| **Precondition** | The mobile is user is in the game lobby.  A game is not currently in progress. |
| **Primary Actor** | Spectator |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. The user selects an option (checkbox) to opt-out of robot control. 2. The user is identified in the waiting lobby with a “(Spectator)” tag next to their user name. 3. The user is moved to the bottom of the lobby waiting list. 4. When a game is launched, the user is not considered for pairing to a remote robot. |
| **Post Condition** | A spectator will never be placed in control of one of the remote robots. |

|  |  |
| --- | --- |
| **Use Case Name** | **Select game type** |
| **Description** | A player selects the type of game to play prior to starting a new game. |
| **Precondition** | The player is in the game lobby.  The player has not opted-out of robot control (not a spectator).  A game is not currently in progress. |
| **Primary Actor** | Player |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. The player is presented with a selection (drop down box) of all supported game modes (“Tank Simulation” and “Light Cycles”). 2. The player selects one of the provided game modes. 3. The selected game mode is displayed in the waiting lobby to all connected users. |
| **Post Condition** | When a game is launched, the selected game mode will be used unless another selection is made before the game begins. |

|  |  |
| --- | --- |
| **Use Case Name** | **Pair player to robot for remote control** |
| **Description** | A specific player is paired to a specific robot for the duration of a single game. Any commands from the player will control only the robot paired to the specific player. |
| **Precondition** | A currently unused robot and unpaired player must be connected to the system.  A game is not in progress. |
| **Primary Actor** | Player |
| **Secondary Actors** | Robot |
| **Dependencies** | None |
| **Basic Flow** | 1. The system selects the first player in the waiting lobby to be paired to a robot. 2. The system selects the first available robot to be paired to the player. 3. The system flags the robot as “in use” (only one player can control a robot) 4. The system moves the paired player to the bottom of the waiting list of players (provides rotation of robot use when more players than robots are connected). |
| **Post Condition** | All further commands from the paired player are propagated only to the paired robot. |

**Virtual Gameplay Use Cases**

|  |  |
| --- | --- |
| **Use Case Name** | **Launch game** |
| **Description** | A player in the lobby initiates a game. The lobby is replaced with an augmented video feed, and players are paired to robots for remote control. Virtual gameplay begins. |
| **Precondition** | The server software is running.  A game must not be in progress.  At least two players must be in the server lobby.  At least two robots must be registered with the system.  A game type has been selected.  A valid camera must have been previously configured for use with the system. |
| **Primary Actor** | Player |
| **Secondary Actors** | Spectator, Robot |
| **Dependencies** | INCLUDE: **Pair player to robot for remote control**  INCLUDE: **Display augmented video** |
| **Basic Flow** | 1. A player selects a “begin game” option from the system’s server lobby. 2. Two or more players are paired to robots for remote control depending on the number required by the game type (INCLUDE USE CASE **Pair player to robot for remote control**). 3. An augmented video feed is displayed to all connected mobile users (INCLUDE USE CASE **Display augmented video**). 4. The system changes state to reflect that a game is in progress. 5. The system starts real time virtual simulation of gameplay and concurrently activates remote control of robots by remote users. |
| **Post Condition** | At least two players are paired to robots.  Robot remote control is activated.  Virtual gameplay simulation is in progress. |

|  |  |
| --- | --- |
| **Use Case Name** | **Display augmented video** |
| **Description** | The live video feed of the camera overlooking the robot arena is combined with the internal game state representation to produce virtual elements overlaid on the live video feed. |
| **Precondition** | The server software is running.  A valid camera must have been previously configured for use with the system. |
| **Primary Actor** | None |
| **Secondary Actors** | Mobile User |
| **Dependencies** | None |
| **Basic Flow** | 1. The system selects a video feed based on the previously configured camera settings. 2. The system renders virtual elements represented within the game state (such as name tags for the robots, any virtual obstacles in play, virtual projectiles, and UI elements such as health bars). 3. The virtually rendered elements are overlaid on the live video feed. 4. The combination of live video and virtually rendered elements are displayed to all connected mobile clients. |
| **Post Condition** | The augmented video feed is visible to all connected mobile clients. |

|  |  |
| --- | --- |
| **Use Case Name** | **Move robot** |
| **Description** | A mobile player issues a command for a robot to move, which is passed to the robot by the system. |
| **Precondition** | The player is currently paired to a robot.  A game is in progress. |
| **Primary Actor** | Player |
| **Secondary Actors** | Robot |
| **Dependencies** | None |
| **Basic Flow** | 1. The player issues a command for a robot to move (either through tilt based controls or an on screen interface). 2. The command is passed to the system through standard internet protocol. 3. The system identifies the robot paired to the specific player. 4. The movement command is transmitted to the robot through the Bluetooth connection. 5. The robot receives the movement command and performs the specified movement. |
| **Post Condition** | The movement command issued by the player is carried out by the paired robot. |

|  |  |
| --- | --- |
| **Use Case Name** | **Fire virtual projectile** |
| **Description** | A mobile player issues a command to fire a virtual projectile, which is created and simulated in the system. |
| **Precondition** | A game is in progress.  The player is currently paired to a robot.  The active game mode is “Tank Simulation”. |
| **Primary Actor** | Player |
| **Secondary Actors** | None |
| **Dependencies** | None |
| **Basic Flow** | 1. The player issues a command to fire a virtual projectile. 2. The command is passed to the system through standard internet protocol. 3. The system identifies the robot paired to the specific player. 4. The system determines the current position and heading of the paired robot. 5. The system creates and simulates a new virtual projectile with the same position and heading. |
| **Post Condition** | A virtual projectile is simulated originating from the player’s paired robot’s position. |

|  |  |
| --- | --- |
| **Use Case Name** | **End game** |
| **Description** | A game in progress is ended and all connected mobile users are returned to the game lobby.  Any paired robots are freed and considered unpaired. |
| **Precondition** | A game is in progress. |
| **Primary Actor** | Player |
| **Secondary Actors** | Robot, Mobile User |
| **Dependencies** | None |
| **Basic Flow** | 1. The end of a game is triggered when directly requested by a player, when connection with an active robot is lost, or when the ending conditions of the active game type are met. 2. The system displays a win/loss message to active players if appropriate. 3. Remote control of robots is disabled (all pairing between robots and players is removed). 4. The system directs the robots to their original starting positions. 5. The augmented video feed to mobile clients is disabled. 6. The system changes state to reflect that no game is currently in progress. 7. The system clears all virtual game state is to prepare for the next game. 8. The server lobby is displayed to all connected mobile clients. |
| **Post Condition** | A game is no longer in progress.  The server lobby is visible to all connected mobile users.  All robots are free (not paired to any player). |

# 

# 

# Appendix C – Use Case Realizations (Sequence Diagrams)











# 

# Appendix D – Class Diagrams

**Note:** All class diagrams have been submitted under the “docs” folder of the software package.

## D.1 – Server-Side Overall Class Diagram



## D.2 – Detailed Controller Class Diagram



## D.3 – Detailed Model Class Diagram



## D.4 – Robot Class Diagram



## D.5 – Android Client Class Diagram

