A Flexible Infrastructure for Multi-Agent Systems

Gerrit Addison N Sorensen
Brigham Young University - Provo

A FLEXIBLE INFRASTRUCTURE FOR MULTI-AGENT SYSTEMS

by

Gerrit A.N. Sorensen

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

August 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Gerrit A.N. Sorensen

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	James K. Archibald, Chair
Date	Randal W. Beard
Date	D.J. Lee

ABSTRACT

A FLEXIBLE INFRASTRUCTURE FOR MULTI-AGENT SYSTEMS

Gerrit A.N. Sorensen

Department of Electrical and Computer Engineering

Master of Science

Multi-Agent coordination and control has been studied for a long time, but has recently gained more interest because of technology improvements allowing smaller, more versatile robots and other types of agents. To facilitate multi-agent experiments between heterogeneous agents, including robots and UAVs, we have created a test-bed with both simulation and hardware capabilities. This thesis discusses the creation of this unique, versatile test-bed for multi-agent experiments, also a unique graph creation algorithm, and some experimental results obtained using the test-bed.

ACKNOWLEDGMENTS

I would like to thank Dr. James Archibald and Dr. Randy Beard for all their help, both scholarly and otherwise, and for giving me the opportunity to work in the MAGICC lab with the many talented students there. Thanks also to Dr. DJ Lee for being on my thesis committee and for his feedback.

I also would like to thank my wife Laura for sticking around through the long period of my student life, and for her help in editing and proofreading this thesis.

I also want to thank Matt Blake and Jeff Nelson for their work on the test-bed, as well as all the senior project students who contributed. To anyone I missed, I am sorry, I didn't mean to, I am grateful to you too.

Provo, Utah Gerrit Sorensen October 29, 2004

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Problem Statement

Research interest in multi-agent systems continues to grow as they are considered for an increasing number of important applications. One area of emphasis in our research in the BYU MAGICC Lab is the design of heterogeneous multi-agent systems for applications in an urban or indoor environment.[1, 2]

A recent electrical engineering graduate class focused on designing and programming a team of five robots to compete in the international Robocup Robot Soccer competition. The class generated ten omnidirectional robots and spurred a desire for further study in the area of multi-agent coordination. The Robot Soccer competition did not provide the full complexity we desired, and so we decided to search for an alternate multi-agent problem that better matched our interests.

After looking at a few options in use at other schools, we decided to define a new multi-agent test-bed problem — a multi-robot version of the game of capture-the-flag. Teams with ground-based robots, a UAV, and a human operator must coordinate their actions as they navigate the maze, attempt to capture the opponent's flag, and defend against robots that have invaded their home territory. For a single human to direct the actions of multiple robots, high-level directives must be supported. In the context of our game, many useful directives would require that each robot have the ability to navigate autonomously through the maze to a desired goal. Thus, path planning is an essential building block in supporting the system functionality that we sought.

To play the game required the design and creation of a fairly large infrastructure including real-time simulation capabilities and hardware and communications systems. This infrastructure, which we call the MAGICC test-bed, required a significant investment in time and effort to create, and went through several incarnations before it reached its current and final state. While originally designed to be used solely in the capture-the-flag setting, the test-bed has evolved into a flexible multi-agent experiment platform that supports experiments in a variety of research areas, including human factors, neglect tolerance, and agent autonomy. A particular strength of our test-bed is that to the agents, there is no difference between simulation and hardware — the test-bed looks the same in either case. This allows the user to test agent code in simulation and then verify it in hardware without making any changes to the software. This flexibility makes the test-bed well suited to facilitate the short turn-around needed in today's fast paced research environment.

1.2 Motivation

The structure of the test-bed changed over time as we ran into problems or discovered better solutions. The discussion on the test-bed divides the design into three separate phases, each of which contains major modifications to some part of the test-bed. While we did not actually explicitly divide our work into "phases" as such, I believe doing so in this thesis provides a useful function. As a thesis on the architectural design of a substantial software system, it provides a look at the design tradeoffs made, based on our assumptions and capabilities during each phase. This hopefully will prove a valuable contribution to anyone designing a similar system, and assist them in making initial design decisions. As the initial design decisions have significant impact on the overall design cost and end functionality, I believe the discussion of the test-bed's evolution provides some valuable design insights, which would be lost if I were to discuss only the final system.

1.3 Outline

This thesis discusses the design and implementation of the MAGICC test-bed through its various iterations, focusing particularly on the various design choices made, and the design lessons learned in each stage. The final design is presented, along with some measurements of its capabilities, and a discussion of its unique aspects and strengths.

The second part of this thesis will focus on the safe-path graph generation algorithm designed specifically to work in a congested urban environment. The attributes of a good path planner are laid out, and some alternate graph generation techniques are discussed. The algorithm itself is discussed in detail, along with a path smoothing algorithm that increases the effectiveness of the graph after creation. The comparative advantages of safe-path over the two alternate techniques previously used in the MAGICC Lab are discussed as well.

Finally, as the value of an experimental test-bed is best established by demonstration, chapter 4 presents some studies done using the test-bed.

Chapter 2

Development of the MAGICC Test-bed

2.1 Introduction

Although the MAGICC Lab has existed for some time, its efforts to create an integrated suite of tools for the development of multi-agent systems have had limited success. Jed Kelsey's thesis [3] focused on the creation of a software toolbox designed to facilitate the coordination and control of multiple robotic agents, and other students have developed additional infrastructure as well. Despite this, students running experiments generally developed their own infrastructure on an ad hoc basis, and although some of the more useful pieces remained, most such infrastructure disappeared after the developing student's graduation. As the development of supporting infrastructure for an experiment often requires more time than running the actual experiment, this sort of development paradigm can involve a large amount of wasted time and effort repeating work that others have already done. While the concept of the MAGICC lab test-bed was originally designed for use in a Senior Project design class focused on the capture-the-flag game, we realized that having a solid, stable and flexible set of development tools would help the increase the lab's research efficiency by reducing the overhead required to create new experiments.

While the test-bed still focuses on the MAGICC lab capture-the-flag game, it has evolved from being useable only in the capture-the-flag multi-agent game setting into a more flexible, multiple experiment platform. This chapter explores the history, requirements, development process, and tradeoffs of the MAGICC test-bed. It focuses on the various design phases the test-bed went through, and on the reasons for the various design decisions made. Each design phase section contains a subsection describing the changes or design decisions made during the design of a particular module. At the end of each design phase section, I discuss the problems encountered, and the advantages and disadvantages of the design during that phase.

2.1.1 Background

The original design of the test-bed was created as part of a Senior Project Capture-the-Flag class for Winter semester 2003. Matt Blake and I came up with a structure, and then I partitioned a team of 12 students to take on various aspects of the test-bed and assist in the design and coding. Matt took the on the task of debugging and perfecting the communication software (called MCF), which he had designed the previous summer but hadn't fully tested, and I worked on the infrastructure. Once we finished the infrastructure the students were to divide into teams of four and use the test-bed in a capture-the-flag competition using agents the students programmed themselves. When we started work on the test-bed, we had only the control for the omni-directional robots, some buggy code for a Voronoi path-planner, and MCF. Despite this lack of initial infrastructure, the students set enthusiastically to work on their parts.

The problem of getting a working test-bed turned out to be bigger than we first thought, and the test-bed was not fully functional before the end of the class. We did have a strong base to work with however, and over the summer of 2003 we worked out most of the bugs in the original code, while refining and changing the overall structure of the test-bed. By fall of 2003, the test-bed was stable, solid, and fast enough to run experiments. However, path-planning remained a major problem. One of the student teams had developed a working $A*$ algorithm, and over the summer we also implemented a Voronoi algorithm designed for use by UAVs, but both had drawbacks when used in the congested maze setting used in the lab. Because of this, I developed the safe-path graph generation algorithm during September-October of 2003. This algorithm facilitated several experiments over the next several months, and became the standard path planning algorithm used with the test-bed because of its stability, speed and path quality. I have since revised the original version to operate on convex polygonal obstacles.

2.2 Capture The Flag

The descriptions of the test-bed and its design in the following sections assume some familiarity with the rules and underlying structure of the capture-the-flag game played in the MAGICC lab. This section explains our reasons for choosing capture-the-flag over other games, and should assist in understanding the decisions made in the design of the test-bed.

As mentioned previously, the test-bed was partially inspired by an electrical engineering graduate class dealing with the creation of a team for the international Robocup competition. After the class we had a simulator and 10 robots, but little of the other infrastructure necessary for the competition. In addition, Robocup has been running for several years now, and many teams are highly advanced, and we felt it would take too long for us to compete on a world-class level. For these reasons we looked at other options that suited our research interests, while allowing us to work on new research.

2.2.1 Overview of the Game

We play a modified version of capture-the-flag, using multiple small, omnidirectional robots, on a field approximately five meters square. The field contains a maze made of rectangular wooden blocks. The game involves two teams, each attempting to capture the other team's flag and return it to a specified location, while simultaneously preventing the opposing team from doing the same. Robots in the opposing team's area can be tagged by the enemy's robots and forced to return to their own side before continuing the game.

Our teams consist of 3 or more robots with simulated sonar (which acts similarly to laser range finding), a simulated UAV flying over the field, and a human operator controlling the robotic agents. The UAV flies over the field locating enemy flags and robots and relaying that information to its teammates. The ground robots must navigate the obstacles on the field while avoiding enemy robots, and attempt to pick up the enemy flag and return it their own side. The human coordinates the actions of the UAV and robots and balances the needs of defense and offense as the game progresses.

A team returning the enemy flag to its own side scores one point. The game continues until one team reaches a pre-decided score, at which point it wins the game. The game moves fairly rapidly, generally taking about 10 minutes for one team to reach 3 points. Capture-the-flag requires good resource management on the user's part, and the team with the better coordination strategy usually wins. Capture-the-flag provides excellent opportunities to study various aspects of multi-agent interaction, including neglect tolerance, human factors, and coordination and control of heterogenous agents.

2.2.2 Definitions

Throughout this thesis I use certain terms, which I define here to prevent confusion. The term 'agent' denotes the software running on a PC containing all the high-level robot control code. The agent code makes all decisions, computes paths, processes messages, tracks positions, etc. Two types of agent code exist: UAV and robot; both function slightly differently. In cases where they differ I will explicitly note which type I am discussing, otherwise both types behave the same. The term 'robot' refers to the actual physical robot itself, which contains only enough software to process messages and control motor voltages. In a sense, these terms 'robot' and 'agent' are inseparable, and with more advanced hardware the robot might run the agent code itself, in which case the two terms would refer to the same object. In our case, because the physical robot does not contain its own control code I will refer to them as separate objects.

The term 'basestation' refers to a GUI through which a user can monitor the action on the field and direct the robots. Each team will have at most one basestation, but no basestation need exist for agents to run.

Figure 2.1 shows the layout of the field, and the names of the respective areas. In terms of location on the field, each team's basestation is considered located in the center of its respective baseline . The team defense area includes its respective untag zone. To get untagged, a robot must move into the untag zone before issuing an untag request to the Referee. Robots in the no-man's-land section of the field cannot tag each other. Team 0 always starts at the top of the field, and team 1 always starts at the bottom. For our lab setup the field width and length are equal, but for simulation the field dimensions can be changed by editing the value in the map configuration file. The x and y coordinates are in millimeters; we use $4500mm$ x $4500mm$ as the standard dimensions for the field. The origin is always located in the upper left-hand corner of the field.

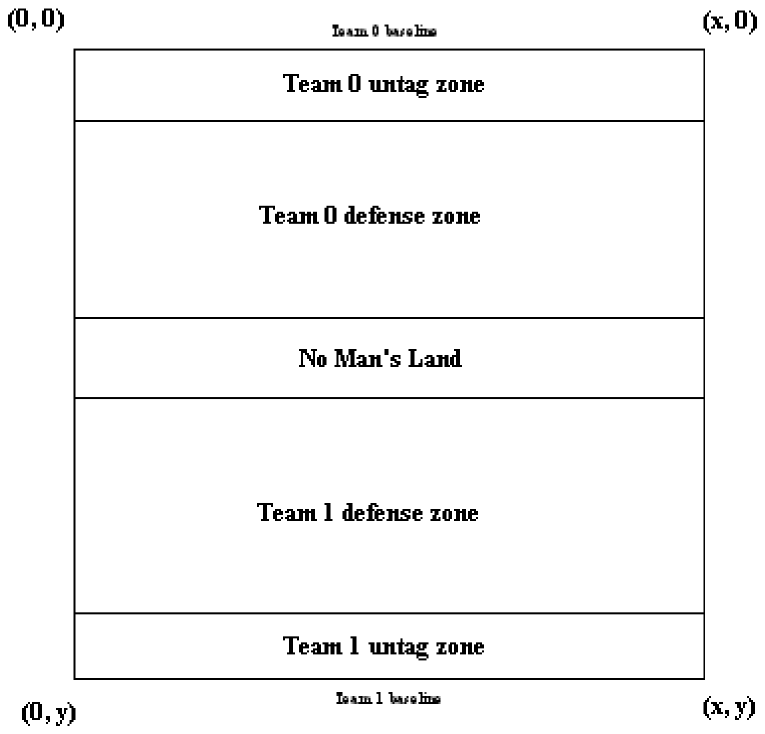


Figure 2.1: The Capture-the-Flag field.

2.2.3 RoboFlag SURF

Before beginning our work on the test-bed we examined implementations of capture-the-flag style games done by other universities. Of all the setups we looked at, one created jointly by Cornell University and the California Institute of Technology called RoboFlag SURF[4, 5] interested us most. RoboFlag consists of two teams playing a version of capture the flag. Each team has an area to defend, and an area to attack. The field consists of a large open square, with a safe area for each team in opposing corners, and the area that each team must defend in the middle of their zone. In addition to the robots, some number of neutral obstacles move randomly about the field. Each team must contend with limited communications bandwidth and limited fuel resources. The teams play two halves, with a break in between. The team with the most points at the end of the second half wins the game.

Robots must move about the field avoiding obstacles, inactive robots, and enemy robots. Robots tag opposing team members by hitting them with golf balls that are placed randomly on the field at the start of the game. Tagged robots must go dormant until the end of the game, becoming obstacles the other robots must avoid. Any robot contacting one of these dormant robots, or one of the neutral obstacles also becomes 'inactive'. In addition, robots which run out of fuel also become dormant. Robots inactivated during the first half are reactivated for the second half.

Teams score when a robot enters the opposing team's flag area and returns to its own flag area without being tagged. Flags are completely symbolic and are 'picked up' simply by entering the opposing team's flag area. Teams can also score points by returning their own flag by tagging an enemy 'carrying' it, tagging enemy robots in their defense zone, and for any inactive enemy robots, including those inactivated by fuel loss and collision with neutral obstacles. This setup provides numerous opportunities for team coordination studies, as well as interface and human factors studies.

Though RoboFlag SURF provides numerous research opportunities, after some study we decided there were some areas in which RoboFlag placed less emphasis, particularly path-planning in an urban-style environment, that particularly interested us. Since the RoboFlag field contained no stationary obstacles to begin the game, it lacked the sort of congested urban environment that we wanted to study. In addition we considered the lack of actual physical flags in RoboFlag to remove additional intriguing research possibilities. Finally, we wanted to study the coordination of ground and air vehicles by adding UAVs to the game to 'fly' over the field and communicate the locations of enemy robots and flags to its team members. We liked the added complexity of forcing the nearly blind ground robots to coordinate their actions with the overhead UAV. Due to these considerations, we decided to create a capture-the-flag style game following a set of rules developed for a congested urban environment, that more particularly suited our areas of interest.

2.2.4 MAGICC Flag

Before beginning any infrastructure design we outlined the game rules and parameters. First, we specified that the game would take place in a maze-like field created using wooden blocks. Such a reconfigurable maze, allows numerous opportunities to study path-planning in actual hardware situations. Having a maze also adds an interesting twist to any team-coordination studies. We also wanted to have physical flags. To prevent game-stopping behaviors like flag-guarding, we decided to place several 'fake' flags, that robots could only distinguish as such when they attempted to pick them up. This forces the defending robots to guard several flags, since even the defenders do not know which flag is the real one. Originally the flags were cardboard, with a metal 'real' flag. We equipped the robots with magnetic servo arms for picking up the real one. The magnets on the servo arm had to be extra strong to pick up the metal flag, and this wreaked havoc with the sensitive onboard computing equipment of the robots. After several failed attempts at shielding the robots from the magnets, we finally scrapped the idea of real flags in favor of virtual flags. We required the robots to mimic the physical behavior required to pick up the flag, without having to have real flags, allowing us to remove the magnets.

This provided several advantages, without significantly affecting the way the game played. First, the physical flags presented a serious challenge to the vision system. The resolution and color differentiation capabilities of the single camera system made it very difficult to track the flags, and impossible to track one that had been picked up. In addition, since the real flag was physically different from the fake flags, if a robot was tagged while holding it, the robots immediately knew the location of the real flag, negating the need to guard multiple flags. Virtual flags removed the involvement of the vision system completely, simplifying the use of flags considerably. Having virtual flags allowed us to change the locations of the flags and add or remove flags as desired. It also allowed us to enter the starting locations of the flags into the map configuration file, simplifying start up procedures. Finally, it allowed us to randomly change which flag is the 'real' flag if a robot happens to get tagged while carrying it, forcing the defense to continue defending all flags.

We also changed the method in which robots tagged each other from that of RoboFlag SURF. In a maze-like environment, it did not make sense to have the robots shooting golf balls at each other, as the obstacles would block most such shots. Instead we required a robot to close within a certain distance of an enemy robot, and then issue a 'tag' request to the Referee. This forces the robots to use their simulated sonar to locate enemies and avoid (or follow) them, or to coordinate somehow with the UAV. The Referee checks the distance between the robots, and if sufficiently close issues a 'tagged' command to the tagged robot. The tagged robot loses the ability to communicate with team members, pick up flags, or tag enemy robots until it returns to its own 'untag zone' and request the Referee to release it. Once untagged, a robot can act normally.

To increase the complexity and depth of possible studies, we added the possibility of constrained communications. Each robot has a variable radius of communication, defined by the user, and enforced by the test-bed. In order to communicate with a teammate, an agent must remain within that radius. Different agent types have different communication radii, for example the base station has a larger radius than the robots, and the UAV can always communicate with the base station, but can only communicate with those robots it can see. The constrained communication environment forces agents to form ad hoc communication networks, and have some way of acting on their own when they lose communication. The test-bed allows the players to turn the constrained communication option on or off as desired for their game.

As mentioned previously, a typical game involves three robots, a base station and a UAV. The robots have only sonar to guide them, and must rely on the UAV to find the flags and relay their location to the attacking robots. The base station acts as a GUI through which the user follows the game, and commands the other team members. The user generally selects one or two robots to go into the enemy defense zone and attempt to pick up the enemy flag, and assigns the remainder to defense. Depending on the sophistication of the base station and the agents, the human player will need more or less involvement in selecting robot destinations, picking up flags, avoiding enemies, tagging enemies, and untagging tagged robots.

Play continues until one side finds the real enemy flag, picks it up and returns it to its own defense zone. The returning side receives one point for returning the flag. In simulation mode the test-bed places the robots and flags back at their starting locations and the game resumes. If playing with actual hardware robots, the robots must be physically moved back into place before the game continues. The players must agree to a score at which to stop play; the Referee will allow the game to continue indefinitely, regardless of score.

2.3 MCF

MCF[6, 7] is a communications software package designed by Matthew Blake of the MAGICC Lab. It is the underlying communications software used in all modules of the test-bed. MCF allows the test-bed to handle an arbitrary number of robots, UAVs, and base-stations, running on arbitrarily chosen machines, without reduction in system performance, and without *a priori* knowledge by the programmer of which machines will be running which processes. The MCF architecture consists of a server resident on each machine, acting as the contact point for all local processes. The user incorporates an MCF client in the software that connects to the server. The servers connect to each other and pass on information about which clients have connected. In this manner each server has a list containing the location of each client in the network. All inter-client network traffic goes through the servers. A client wanting to communicate with another asks the server for the ID of the client it wishes to communicate with, and then passes all data to the server, along with the receiver's ID. The server sends that data to the server on the host machine of the second client, which passes the data up to the client. This system allows the client to make a single connection to a single server, and still communicate with an arbitrary number of other clients. It forms the backbone of the MAGICC test-bed communication system.

2.4 System Configuration

The original robot soccer simulator included a configuration file format used to configure each team of robots, and the test-bed uses the same format for its robots. The configuration file includes the physical aspects of the robot and can be modified by the user to allow virtually any type of polygonal robot. We extended this concept to include a configuration file for the map, since we wanted the ability to use any number of different maze setups. The map configuration file contains the data for all obstacles on the field; it lists the vertices of each obstacle in Cartesian coordinates. Later we added the location of all the flags, and the size of the field. I rewrote the parser from the original simulator to work with the map configuration file. We place the configuration file in a known location and then any module using the obstacles reads this file using the modified parser object. This format allows us to keep multiple map files, and switch them easily when needed.

In addition to the map configuration file, key game parameters are placed in a header file included in all system components. These parameters include the standard field dimensions, robot tag distance, and robot communication parameters, along with some common classes and programming structures. Changing this file requires recompilation of the code to take effect so it is reserved for parameters that change infrequently.

2.5 Design Phase One

As stated previously, the original design of the test-bed was inspired by the Robocup competition infrastructure, where each team has a single command entity which controls the robots, and makes the decisions for the team. This entity also

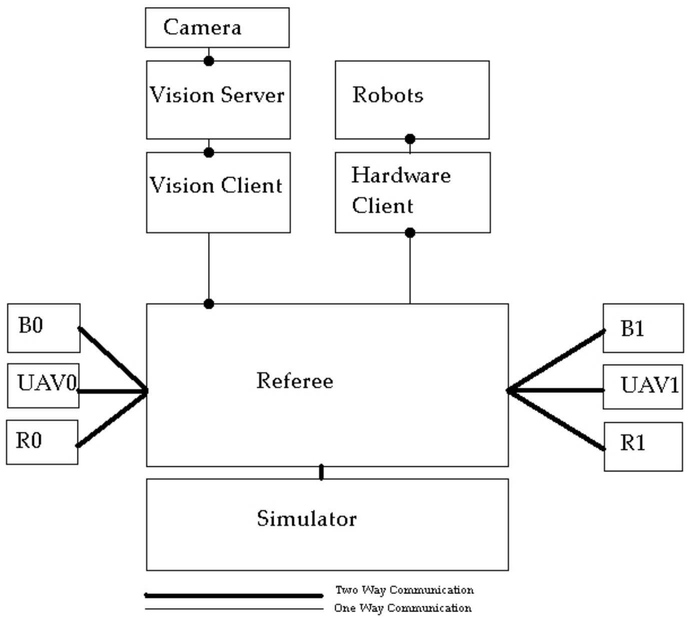


Figure 2.2: A schematic of the original test-bed design. The black dots denote the recipient of one-way communication. The smaller boxes represent the basestation, UAV and robots for each team.

communicates with a game moderator called the Referee, which controls all aspects of the game, including time, scoring and calling fouls. Our design, while following this format to a certain extent, differs both in the number of modules, and in the decentralized nature of the teams. Figure 2.2 shows the complete layout of the original design.

The Referee acts as the central hub through which all other modules communicate. The test-bed runs in two different modes: simulation and hardware, and requires different modules for each mode. In simulation mode the test-bed runs only the simulator and the Referee; robots communicate with the simulator through the Referee, and the simulator provides the vision and sonar data for the system. In hardware mode the vision server receives data from the camera and extracts the vision data. It passes the vision information to the vision client which uses it to create the sonar information for the robots. In hardware mode the robot agents calculate their control values and send them to the hardware client which transmits them to the robots via wireless modem. I explain the individual pieces in the following sections.

2.5.1 FlagClient Communications Framework

In any multi-agent experiment, communication between agents comprises a major practical problem to solve. The problem is non-trivial in both programming requirements and logistics, and can pose a major challenge. As part of our design we wanted to create a framework that simplified the task of communication between agents, and reduce the work required to implement it. We did this by creating an entity we called the FlagClient. This object handles all communications between an agent and its team members and between the agent and the Referee. It creates the MCF client, and registers the agent with appropriate modules, freeing the programmer from these tasks. The programmer using the FlagClient for a particular agent needs only the ID of the agents and modules to communicate with, and then can send and read messages using only these IDs. The agent passes this data to the FlagClient, which handles the actual communication using the MCF client as described in section 2.3. The FlagClient also handles the parsing of messages, passing the sender's ID and the message received to the calling function. The FlagClient removes the burden of programming the registration and communication process, allowing users to focus on the essential parts of their research. By putting all the basic communication functions in one place, this type of modular software design makes modification easier when necessary. The FlagClient was originally written by Ryan Faulk with later modifications by myself and Matt Blake. The FlagClient remained basically the same through all design phases, receiving only minor updates as we changed communication protocols.

2.5.2 Referee

The original version of the Referee functioned as the communication hub for the infrastructure, and the arbiter of the rules. The phase one design had no GUI, and no means of user input after startup. All game parameters, including the number of teams, the number of agents per team, and whether to run in hardware or simulation, were set using command line arguments, and any change to the game parameters required a complete system restart. All communication between modules passed through the Referee, which checked each message and then sent it on to the proper recipient(s). The Referee dispersed all vision data sent by the simulator and the vision client; after updating its own robot position data, it distributed the vision data to the individual robot agents. Agents registered with the Referee, which then assigned each agent an ID to allow the system (including teammates) to differentiate between them. Robot agents received two IDs: one system ID, and one for use with the simulator. The robot agents used their system ID for communicating with teammates and the Referee, and the simulator ID for sending move commands to the simulator. The Referee assigned IDs based on time of registration, starting with zero, and incrementing the ID for each succeeding registration.

To make it easier for users to follow the game, we envisioned the Referee sending the vision data on to a monitor program that would display the robot, UAV, and flag positions on the field, however this program was never written. Instead, the Referee output the positions to the screen textually, which allowed users to track robot positions and monitor the game, albeit with some difficulty.

In the phase one the Referee generally had a separate communication protocol for each module, depending on the type of communication required. For example, all the communication between the robots and Referee used strings to make debugging and reading the code easier, but the Vision Client sent all its data as a data structure that the Referee reassembled on receipt. We did this to make constructing the data packets easier when sending large amounts of data, such as with vision. This had drawbacks, however, and resulted in multiple methods of parsing data, which complicated the code unnecessarily.

Besides handling communication, the Referee tracked tags and untags, communication between robots, flag pickups and drops, and issued an alert when a team successfully returned the enemy flag to its own side. The Referee also registered each agent, tracked its status throughout the game, and handled all UAV movement and vision functions.

The original Referee code structure did not use a timer, as does the simulator. Instead, after reading the command line parameters and using them in the initialization function to create the map and teams, the Referee entered its main loop, which it remained in until shut down by the user. The main loop processed a single message, checked tags, tracked the flags, and then gave up the CPU by calling a sleep function. The message processing format the Referee followed remained basically the same through all design phases: check the sender, and then call a parsing function based on the sender's identity. This was particularly important at the start, because as noted previously, different senders used different message formats, and their messages required different parsing techniques. The message parsing functions read the message and then call other functions as appropriate to handle it.

Such modularization formed an important part of our programming methodology. In such a large project maintaining organized code requires some effort. Giving each task its own function, and minimizing the number of unrelated tasks per function helps to modularize the code within a program. For example, even after standardizing the message formats we kept separate message parsers for agents and infrastructure modules, rather than having a single message parsing function for all messages. This modularity in coding simplifies reading the code, reduces the size of functions, and helps to keep the code logically organized. In addition, it allows reuse of functions by making them more specific to a single task, and makes locating pieces of code easier. We tried to make the code in all parts of the test-bed as modular as possible.

2.5.3 Real-time Simulator

Preston Jackson originally designed the simulator structure for use with the RoboCup five-man robot soccer competition. It features real-time simulation with collision detection, and uses actual robot dimensions as defined by the user. While we kept the basic framework of the original, I rewrote most of the internal parts of the simulator to work with the test-bed. For example, the original simulator used multiple separate threads to listen for incoming TCP connections to different code objects. MCF removed the need for these threads so I removed them and placed the communications from the separate objects into a single dispatcher object. The dispatcher object keeps the MCF IDs of all processes the simulator communicates with and handles all communication with the simulator. It parses incoming messages and formats outgoing messages, and handles robot registration.

At startup the simulator creates the robot objects using a configuration file supplied by the user. The phase one simulator allowed for a fixed number of teams (either one or two) with a fixed number of team members; it could not handle arbitrary sized teams. The user entered these values using command line arguments, and once the simulator started could not change them. The simulator created an array for each team using the values entered on the command line. If the proper number of robots failed to register, or if too many registered, it would give an error and exit. After creating the robots, the simulator created a vector of obstacles using the central map configuration file. It then created the dispatcher object and connected to the Referee. At this point it could accept incoming registrations, and when signalled by the Referee, started its simulation.

2.5.3.1 Simulator Engine

The simulator functions as a timer-activated loop. The timer calls the loop function at a user-defined rate, depending on user requirements and CPU capabilities. While in the loop the simulator handles all communication, responds to Referee commands, calculates robot movement, handles collisions occurring due to movement, calculates the sonar information for each robot, and then sends the vision data out to the robots and the Referee, in that order.

Movement is handled as simply as possible to maximize the speed of the simulator. The simulator keeps track of the velocity and position of each robot at each time step. Each agent sends the simulator its desired velocity, which the simulator maintains until changed by the agent or by collision. Since instantaneous changes in velocity defy realism, the simulator checks each robot's velocity against the maximum velocity and acceleration defined by the user in the configuration file, and sets any invalid changes to velocity at a valid bound. The current velocity of the robot is divided by the number of frames per second the simulator is running at to get the robot's speed in meters per frame. This value is then added to the robot's position at each frame to get the robot's proposed position. The simulator checks for any collisions using the proposed positions, rather than the actual position.

For collision detection purposes, the simulator maintains the two-dimensional physical dimensions of the robot as a list of points, each of which specifies a vertex of the robot's polygonal area, as viewed from directly above. The simulator maintains the list of obstacles in the same format. These points define the line segments that make up the convex hull of the robots and obstacles. The collision detection algorithm checks each line segment in the robot against the line segments of all other robots and obstacles on the field. An intersection means the polygons overlap and a collision has occurred. To reduce the number of (relatively) expensive line intersection checks performed, we check bounding circle[1] intersections first for each polygon. If no bounding circle intersection occurs then the two polygons cannot intersect and the simulator skips the line segment intersection checks. In addition, after detecting a collision, the simulator forgoes any remaining checks, as multiple collisions do not require special treatment. If two robots collide, the simulator does not check the second robot separately, which also speeds up the collision detection. When a collision occurs, the simulator sets the colliding robot's velocity to zero, and leaves the robot in its original position, discarding the proposed position. When no collision occurs (the usual case), it changes the robot's position to the proposed position, and sends the updated information out to the robot. With properly defined robot characteristics this algorithm accurately models real-world robot behavior. The algorithm runs best at a rate of at least 10 frames per second. Higher frame rates produce more realistic results by minimizing the distances traveled between frames.

In addition to providing the vision data, the simulator sends each agent its sonar information along with its position. As the simulator currently functions, the

[1]The minimum area circle containing all vertices of a polygon

robots receive perfect sonar information, but we could easily add uncertainty to the data if desired. The next section describes the algorithm used to create the sonar data.

2.5.3.2 Sonar Detection Algorithm

As previously mentioned, the main sensor for the robots is the simulated sonar. The simulator generates sonar information at every time—step, requiring a fast, accurate algorithm allowing variable numbers of sonar points. The sonar is not true sonar, but actually functions more like a laser range finder. Each robot receives a list of sonar data points as part of its vision data. Each point represents the distance the sonar ‘sees’ from the robot at a certain angle, starting with the angle the robot is currently facing. The sonar algorithm is shown in Algorithm 2.1. The agents are free to use the sonar data in whatever way they wish; the simulator places no restrictions on its use. The vision server uses the same algorithm to generate sonar data when running in hardware mode.

2.5.4 Vision Server and Vision Client

Jeff Anderson designed the original vision server for use with a single camera positioned approximately 15 feet above the field. The camera required a fish-eye lens to see the whole field, which warped the image somewhat, requiring the vision server to de-warp the image before extracting the robot positions. After de-warping the incoming frames from the frame grabber, the vision server derived the robot positions from the image and sent them out as an array of double values in a set order. The receiving client extracted the data from the array as needed. Any number of clients could connect to the vision server and receive vision data: however, in the test-bed only the vision client and the hardware client connected to the vision server.

Adding the sonar data in hardware mode required an intermediate process to receive the vision data from the vision server and use it to calculate the sonar values for each robot. I created the vision client, which performed this function in the phase one design. The vision client received the robot positions from the vision

Algorithm 2.1 Sonar Algorithm

```
1: for numSonarPoints do
2:    if numSonarPoints = 0 then
3:       $theta$ = (current robot heading)
4:    else
5:       $theta = theta + (2 * pi/numSonarPoints)$
6:    end if
7:    Find the line segment where the first endpoint is the center of the robot, and
      the second endpoint is found using the following equations:
      $x_2 = (maxDistance * cos(theta)) + x_1$
      $y_2 = (maxDistance * sin(theta)) + y_1$
      {$maxDistance$ is the range of the sonar, $theta$ is the current sonar angle, $x_1$ and
      $y_1$ are the coordinates of the first endpoint, and $x_2$ and $y_2$ are the coordinates
      of the second endpoint}
8:    Find the intersection point closest to the robot of the line segment created in
      with all other robots and obstacles. This intersection point is the sonar point
      returned with the vision data.
9: end for
```

server and then used the simulator sonar detection algorithm to calculate the sonar values at each robot's position. The vision client then sent this data, encapsulated into a compact structure, to the Referee which extracted the data and sent it to the individual robot agents.

2.5.5 Hardware Client

As mentioned previously, the robots only contain enough processing power to handle serial communication and process some control functions. The hardware client functioned as the intermediary between agents and their respective robots. It received motor voltages from the agents and sent them out over a wireless modem to the hardware robot. As I originally created it, it functioned purely as a translation service, taking incoming packet data, and sending it byte by byte over the wireless modem. The robot agents calculated the control values transmitted by the hardware client. As the control loop for the omnidirectional robots requires a strict 30 frames per second to produce good values, it quickly became apparent that the robot agents could not meet the timing requirements of the control loops because of various delays. To solve this problem, Ryan Faulk revised the system to allow the hardware client to calculate the control values for all the robots.

In the revised version, the agents sent the hardware client their desired velocity in the x and y axes. Using vision data obtained from the Referee, the hardware client calculated the required control values for each robot's motors. After deriving the control values, it sent them out over the serial line to the robots. At first, we sent the data as a single packet containing the control velocities for all the robots on the field. However, this made it difficult to make changes to the hardware, because each robot had to know which position in the packet contained its particular data, and this had to match the top given to the robot. Because of this, we modified the system so the hardware client sent the data out in individual packets, with an identifying header for each robot. The robot looked at the header and ignored any packets not addressed to itself. This made the code simpler, but increased the overhead of data sent via the wireless modem. This increase in overhead in turn increased the amount of time the hardware client spent transmitting data and introduced delays into the control loop when more than a certain number of robots required data. To combat this, we simply limited the number of robots per team to three, which gave the hardware client sufficient time to perform satisfactorily at 30 frames per second.

2.5.6 Design Phase One Conclusion

When we first started the project, we did not fully realize the limitations inherent in running a real-time distributed system on a system not designed to handle real-time operations. For example, while the usleep function provided in the standard C++ library offers granularity in microseconds, the actual PC clock timer has a granularity of 10 ms, making the usleep functionality moot. When we designed our original loops throughout the entire system, we based them on the usleep timing. Since the CPU could not handle that granularity, the loops could not handle enough messages and hogged the CPU, causing further slowdown to the entire system. We fixed this problem during phase two by redesigning the loop timing system used in most processes.

The problems with the timing strategy initially used increased the overloading of the Referee. With a distributed system such as this, where the functioning of one process depends on the receipt of data from another, overloading a single process slows the whole system down. In design phase one, the amount of message traffic quickly caused the referee to become a bottleneck, slowing the entire system down and causing errors due to dropped messages. This problem was especially acute for the messages passed between the vision server and the robots, as any significant delay caused huge errors in the motor control functions, making the robots virtually impossible to control. Besides handling all message traffic, the Referee simulated the UAV movement and sight, which further increased the code size of the Referee and slowed it down. To reduce the complexity of the Referee and increase its speed we removed some of its functions and placed them in their own dedicated modules.

If the system were to run in simulation only, our initial structure would not have been so problematic, however the control of real-life hardware robots is significantly more sensitive to delay than simulation. Delays in the simulator do not change its behavior significantly, while even small delays in the hardware control loop can seriously degrade its functionality. Because of the delays introduced by our initial structure we had very poor performance in hardware mode. Design phase two focused on restructuring the system to allow the hardware to perform as expected as well as creating new functionality to facilitate the ease of using the system.

2.6 Design Phase Two

Design phase two focused on restructuring the system to remove delay and speed up the message transmission. The Referee received the largest overhaul, including the addition of a GUI to allow for visual monitoring, and the removal of the UAV simulation tools into a separate module. We changed the timing mechanism of most modules, and updated them to work with the new Referee GUI functions. Finally, we reworked the hardware systems to take advantage of new modem technology, and to increase the number of robots allowed on the field. Figure 2.3 shows the infrastructure during design phase two. Other than the addition of the UAVmodule, and an additional hardware client, the basic structure of the infrastructure remained the same.

2.6.1 Referee

As mentioned previously, we found it necessary to change the message handling so that the Referee handled messages between agents only, allowing the agents to communicate directly with other modules. This change in message handling reduced the load on the Referee, allowed the Referee to handle all the messages between robots without dropping messages, and reduced the overall error rate of the system. We did this by having the agents register with each module separately. This did not complicate things significantly, as each process already kept the communication IDs of all modules it communicated with, and so required little new information. Due

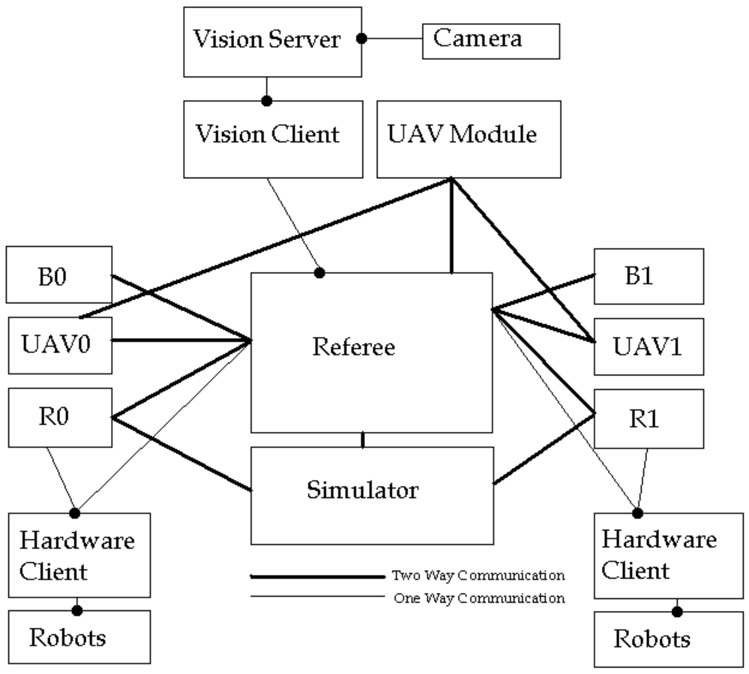


Figure 2.3: The infrastructure after design phase two.

to the modularity of the code, adding registration functions to each module required little more than cut-and-paste.

During this phase, we also changed the Referee's structure to allow for more specific definition of the timing. In the previous phase we relied on the sleep function to provide timing, but as noted previously, this hogged CPU cycles because the loops never exited. To fix this we implemented a timer that called the main loop at specific intervals. We used the native QT QTimer class because of the computationally optimal timing capabilities it provides. This configuration allows us to change the timer frequencies, depending on the capabilities of the host machine and the network. The timer calls the loop, which exits after running, freeing the CPU for other processes. The Referee loop itself is quite simple and is shown in algorithm 2.2. Notice that in this version the Referee processes **all** messages in the queue at each time step, rather than processing a single message as before. This significantly increases the number of messages the Referee can process per second, and reduces the number of dropped messages to a negligible number under normal operating conditions.

Algorithm 2.2 Referee Processing Loop

```
while messages in queue do
  if message from simulator or vision server then
    process vision
    handle flags
  else if message from uavmodule then
    process UAV data
  else if message from agent then
    process agent message
  end if
end while
```

All game functions are called from within one of the message parsing functions in response to a particular message. Originally we had the Referee make certain checks, such as tags, every time through the loop, but this wasted computation time because tags occur relatively rarely. In the general case the Referee runs faster by running checks only when it receives a request. For example the Referee does not check for tags with every loop, but only when a particular robot requests it. This reduces the computation required to process requests as long as agents keep the number of requests within a reasonable bound. If all agents constantly send a tag request even when they are nowhere near an enemy, it could overflow the Referee's message queue, causing dropped messages. However, since most game requests happen on an infrequent basis, and because the time required to process a message is much less than the time per loop iteration, this approach works well in most situations. Increasing the number of agents increases the burden on the Referee, but reducing the number of iterations per second the agents execute will reduce the number of messages sent. Reducing the rate at which the Referee runs helps also, as it causes fewer context switches, allowing it to use more CPU cycles. We have tested the system with 32 robot agents at 10 frames per second without losing messages due to buffer overflow. We tested this by having each robot send a certain number of messages, and count the number received; if they match then no messages were dropped during transit.

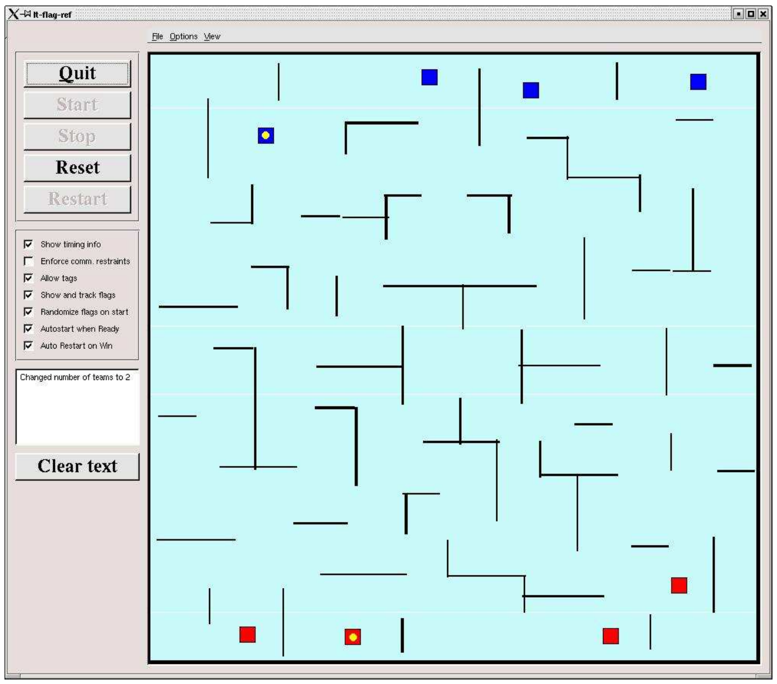
The main drawback of this type of loop is that there is no bound on the time each iteration takes, except the size of the message buffer. Since the Referee must process all messages in its queue before exiting the loop, there is no way to guarantee the Referee meets any timing requirement. The timer calls the loop at a specific rate, but it cannot force the loop to exit to maintain that rate. With a large message load at higher frame rates, the loop can go over its allotted time, dropping the frame rate. In our system, only the display of the GUI depends on the Referee maintaining a certain frame rate so decreasing the frame rate can make the movement on the display seem somewhat jerky. All other modules have similar timing loops and thus can maintain their own timing as necessary. Our timing strategy provides consistent timing as long as the computation time required to process all the messages in the queue is less than the time-per-frame. In normal circumstances the Referee maintains the desired frame rate with less than 1% error.

In phase one the Referee provided no visual feedback on the action happening on the field, which made the game hard to follow and made it more difficult to debug the other modules. To overcome this, I incorporated the original Referee code into a graphical user interface (GUI) using QT. The Referee now serves as a visual, interactive means of controlling the game, with the ability to stop, start, reset, or quit the game at the press of a button. In addition the GUI allows a user to view and manipulate the obstacles on the field, add or delete obstacles and flags, and move robots around the field (in simulation mode) using a mouse. The GUI provides the means to change the number of teams playing the game, load new map files, and select different game options such as enforcing communications restraints, without restarting the Referee or other modules such as the simulator. The Referee communicates user commands and changes in game parameters to all modules connected to it, allowing the user to modify the game setup from a single convenient point.

Users issue the stop, start, restart, reset, and quit commands by pressing buttons on the main interface. These commands allow the Referee to stop or start all modules connected to it, reset the game by disconnecting all agents, and exit the entire system by closing all modules at once. Before the GUI was implemented, the game parameters were set using the command line, and changing them or starting a new game required a complete system shutdown. The GUI allows the user to change parameters such as the number of teams without shutting down any modules. In addition, starting a new game requires only a reset to disconnect all agents, and then new agents can register as normal. Stopping the game allows a user to move robots around the field, change flag locations, or perform other tasks such as modifying the maze, while preventing the robots from moving or taking other actions. Restarting the game returns the robots to their original starting locations, resets the score, and randomizes which flag is the real flag. Figure 2.4 shows the Referee GUI with no robots registered. The colored squares show the locations of the flags; the real flag for each team has a yellow dot in the center.

2.6.1.1 Communication Algorithm

One of our goals for inter-agent communication was to allow the enforcement of limited communication distances between robots. The communication rules defined for the test-bed allow a robot to communicate with any teammate within a specified radius, or indirectly with others as long as each transmission step is in range. This format forces a sort of ad hoc network to be formed, wherein robots must maintain their links or be cut off from communication. The original algorithm, which checked communication constraints, cross-referenced two arrays and had a worst case execution time of $O(n^4)$, where n is the number of robots. This proved too slow with teams of more than 4 robots, and so I wrote a new recursive algorithm that runs in $O(n \log n)$. The algorithm begins with the original source robot, and checks each



☑ Show timing info
☐ Enforce comm. restraints
☑ Allow tags
☑ Show and track flags
☑ Randomize flags on start
☑ Autostart when Ready
☑ Auto Restart on Win

Changed number of teams to 2

Figure 2.4: The Referee GUI. Each team has a different color.

agent on the team until it finds one within the source's communication radius. If that agent is the destination agent the algorithm returns a boolean true result, otherwise the agent is placed into a list of robots the original source can communicate with, and the algorithm starts over on the new robot. The algorithm works its way through any existing links to the destination robot, and returns a true value. If no links exist to the destination the algorithm will return a false value. This method does not require the Referee to maintain or update any sort of communication table as the previous method did. However, since it must run every time an agent communicates with another agent, it can result in unnecessary function calls for multiple messages from the same agent. In practice it runs fast enough that it does not cause noticeable slowdown. A possible optimization would be to save a list of agents each agent can communicate with and check it before running the algorithm. The list would remain valid for only a short time before clearing itself.

2.6.2 Real-time Simulator

I modified the simulator in phase two to work with the new Referee game commands built into the GUI: stop, start etc. In addition I changed the simulator to allow arbitrary numbers of robots on a team. The simulator maintains a vector of robot objects, and creates them individually as the agents register, rather than creating the entire team on startup as done previously. This removes the necessity of command line arguments specifying team members, and gives the user much more flexibility in the game scenarios allowed. When the Referee issues a reset command, the simulator clears the robot vector and registers new agents, allowing the simulator to handle multiple, distinct team configurations without rerunning the program.

As part of the communication changes for design phase two, we changed the registration process so that agents register directly with the simulator. The Referee no longer assigns agents a simulator ID; instead, it tracks the order robot agents register with the simulator, and implicitly deduces the simulator ID. The Referee uses the simulator ID of each robot when receiving its vision data and notifies the simulator when the user moves a robot using the GUI. The simulator uses the ID to directly reference robots in the vector. The agents communicate movement commands to and receive vision data directly from the simulator, rather than through the Referee as done previously. This change substantially reduced the message load seen at the Referee, while increasing the simulator message traffic due to the increased overhead incurred by sending vision to multiple recipients. Overall the amount of vision traffic remained constant, but by distributing the load we removed the bottleneck at the Referee.

2.6.3 UAVmodule

I created this module because running the UAV simulation code in the Referee required too much computation time. The UAVmodule took over all simulation duties involving UAVs, including movement and vision. Like the simulator the UAVmodule can handle an arbitrary number of UAVs on each team, and does not need to know the number *a priori*.

To simulate the UAVs' movement I used a constant velocity model; the agent sends only its desired heading changes to the UAVmodule. Algorithm 2.3 shows the method of calculating each UAV's position. Once the UAVmodule calculates each UAV's current position, it uses that data to create a list of all world objects the UAV can see. Each UAV has a square sight 'footprint' rather than a circular one; this simplifies the list creation without significantly changing the amount of the world the UAV can see. UAVs can see robots, flags, and obstacles; the UAVmodule communicates the number and location of each back to the UAV in a human readable string of variable length. The UAVmodule receives robot position data from either the vision server or the simulator depending on the mode; the Referee sends a list of flag positions when ever a robot moves one of the flags.

Algorithm 2.3 UAV Movement

```
1: for Each UAV do
2:     if $\omega_{desired} > \omega_{max}$ then
3:         $\omega_{desired} = \omega_{max}$
4:     end if
5:     $UAVPOS.x+ = VELOCITY * cos\omega_{desired}$
6:     $UAVPOS.y+ = VELOCITY * sin\omega_{desired}$
7:     $UAVPOS.theta+ = \omega_{desired}$
8: end for
```

2.6.4 Vision Client

The vision client remained basically the same during this phase except, like the other modules, we modified it to allow the agents to register directly with it.

The vision client communicated directly with each robot after this phase, registering clients in a manner similar to that of the simulator. After registering an agent, the vision client sent the agent vision data directly rather than through the Referee. In addition I updated the vision client to work with the new Referee game commands: stop, start, restart, reset and quit. Because of the vision system, the vision client remained a command line process with static game parameters. Changing the number of robots on the field required the user to close the vision client and restart with the new parameters. The other modules could continue running while the user restarted the vision client. I also standardized the vision output format to match the simulator's so that clients receiving vision data can process it the same way in both hardware and simulation modes.

2.6.5 Hardware Client

During this phase we acquired some new wireless modems with multi-channel capabilities, so we changed the hardware client to make use of these. Instead of having a single client to handle all the robots, we ran one hardware client for each team. Each hardware client transmitted on its own channel to all the robots on its team. This allowed us to double the maximum number of robots per team, while still maintaining the required 30 frames per second for the control loops. Like the vision client, the hardware clients had to restart when the number of robots on the field changed.

2.6.6 Design Phase Two Conclusion

After restructuring system communications the system ran well enough to handle experiments in both simulation and hardware modes. However, it still had some flaws, particularly in ease of use. Because of the changes to the hardware client, running in hardware mode with two teams required a user to start five modules, not including any agents, and some modules still required static game parameters. The variable starting methods made it inconvenient to run repeated experiments because of the number of processes requiring shutdown and restart between test-runs.

During this phase we began standardizing message formats, particularly the vision data. Having a standard vision format allowed agents to process the vision data the same in both simulation and hardware modes, moving the system towards mode transparency.

Adding the GUI increased the functionality of the system significantly. Without the ability to visually track robot positions, debugging modules (e.g. the simulator, vision client, and basestation) was much more difficult. Before the creation of the GUI users had to adjust game settings separately for each module, often requiring restarting the module. The ability to control the entire system from a central location reduced the complexity of the infrastructure, making it easier to use.

Finally, removing the necessity of statically set game parameters let users run multiple experiments on the same system incarnation without shutting the system down. The original system added a high overhead in the time required between experimental test-runs, because the entire system had to shut down and restart between runs. This overhead made using the system a significant headache.

2.7 Design Phase Three

In phase three we refined the infrastructure somewhat by removing the hardware and vision clients, and replacing them with a new module: the control server. In addition the vision system received a significant upgrade. By this phase the Referee, UAVmodule and simulator functioned well, and changed very little. Figure 2.5 shows the infrastructure after design phase three.

2.7.1 Control Server

The control server replaced the hardware clients and vision client for each team and added control functionality to the system in hardware mode. Although having separate hardware clients for each team improved the hardware performance, the robots still did not behave as well as desired in hardware mode, and the hardware clients did not implement all available control functions. In addition, the large number of modules required for hardware mode made the system unwieldy.

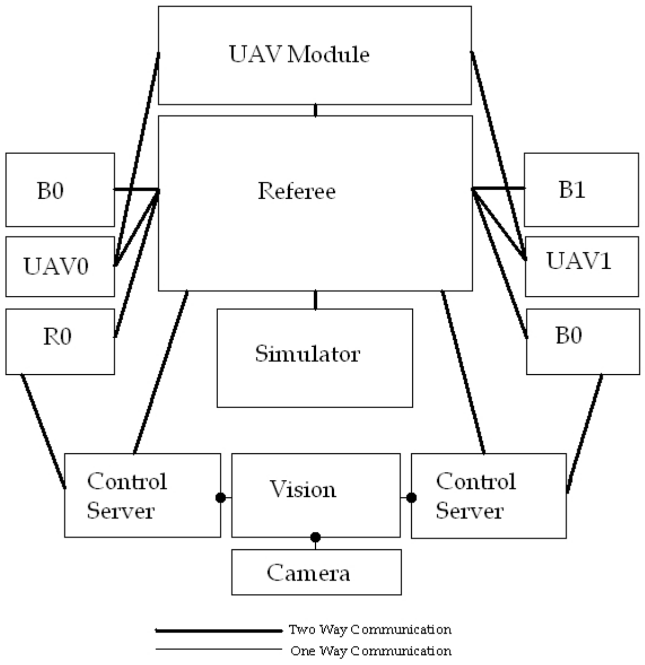


Figure 2.5: The infrastructure after design phase three.

The control server assumed the duties of the vision and hardware clients for each team. Placing the control, vision and sonar functions in one place also removed any synchronization errors introduced by using the vision data in two separate places. Each team had its own control server which connected to the vision system and requested the vision data for its team members. Receiving vision data directly from the vision server rather than from the Referee reduced the delay in the control loops significantly, which in turn improved the response and performance of the robots on the field. This change allowed us to run the robots in hardware mode with similar behavior to the robots in simulation mode, a previously impossible situation.

The robots registered with the control server instead of with the hardware client and vision client, and the control server sent out the vision data to the robots. The control server reduced the number of modules required to run in hardware mode from five to four. This reduction made the system easier to use, particularly when changing the number of robots per team, because only two modules needed restarting instead of three.

The control server implemented additional movement types into the control loop, giving robot agents access to commands such as move-to-point, turn-to-point, follow-line, and maintain-facing. These move commands increased the functionality of the robots in hardware mode, but since they were not available in simulation mode they did not get used often, because they required programming separate move functions for each mode.

2.7.2 Vision Server

As noted previously, the original camera setup contained one camera placed 15 feet above the field, with a fish-eye lens allowing it to view the entire field. This setup, while functional, was not optimal as the fish-eye lens caused warping, which in turn caused position errors, particularly near the edges of the field. This situation was impossible to remedy using a single camera, because of restraints on the placement of the camera. After some study, we determined that using four cameras to view the field would eliminate the need for fish-eye lenses, in addition to increasing the overall resolution of the system. Accordingly, we purchased four new video cameras, which were installed above the field.

The new vision server is composed of four scanning clients, each responsible for a single camera, and the main vision server. Each of the clients scans the images returned by its camera, and extracts the position information of any robots in the image. If two or more cameras see a particular robot, the position information for that robot is averaged from all available data by the main server. The server combines the position data extracted by the individual cameras and distributes it to any connected clients.

Currently only the control server connects to the vision server. It might seem that the need for processing four separate images would make for a slower system, but each image is about 1/4 the size of the image in the original system, and so requires much less processing time. In addition, because each camera is responsible for only 1/4 of the field they can be mounted lower and do not need a fish-eye lens, eliminating warping. Removing the need for the computationally expensive de-warping algorithm significantly increases processing speed, resulting in a speedup in overall vision processing. The increased resolution of the images returned by the cameras allows for more accurate positioning, particularly near the edges of the field. This new vision system was designed and implemented by Matt Blake.

2.7.3 Design Phase Three Conclusion

The control servers served an important purpose in placing all the vision processing in a single location. Having vision data processed in separate locations can cause synchronization errors, and reduces system efficiency through redundant processing. The control servers removed this redundancy. During the previous design phases we overestimated the processing power required for most functions, erroneously attributing poor performance to overloaded CPUs, when in fact the performance problems were caused by the timing strategies originally employed. Once we redesigned the timing system we could remove redundant systems such as the vision and hardware clients and place similar functions in a single process. This had the additional benefit of reducing the network traffic load and system complexity. However, by introducing a new movement interface for hardware mode, the control server moved us farther from our goal of mode transparency by forcing agents to send different move commands depending on the mode.

2.8 Final Design

The schematic for the final design is shown in Figure 2.6. The final phase involved changes to the control server and simulator. We tested the whole system during this phase extensively, and ran it for hours at a time in simulation mode without problems. The system is not quite as robust in hardware mode, but it functions reasonably well as long as the robot battery supplies last. The final system has mode transparency, meaning agent code runs the same in both simulation and

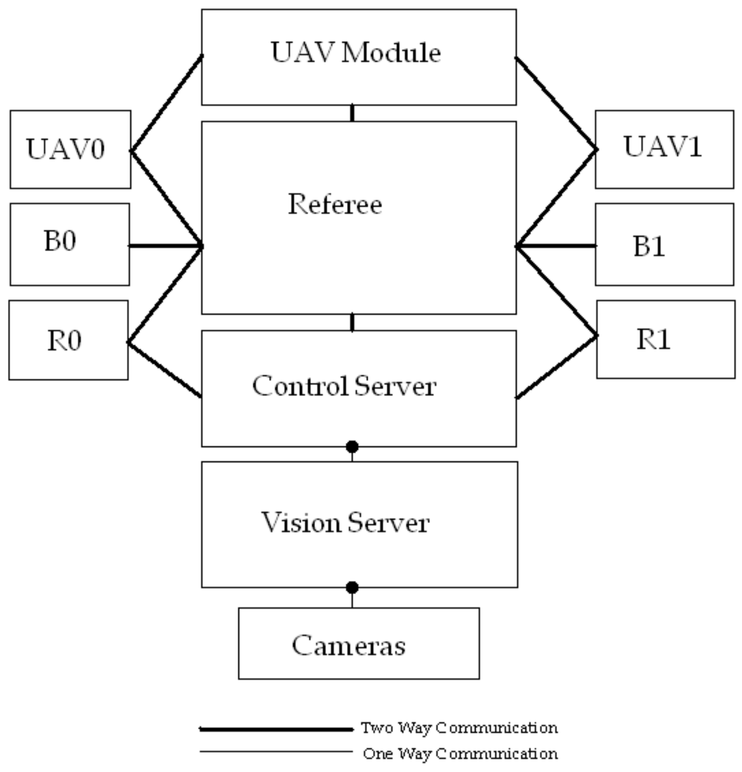


Figure 2.6: A schematic of the final test-bed design.

hardware, so an experiment run in simulation can be verified in hardware without any modification to the code.

2.8.1 Control Server

One of the goals we had was to make the system look the same to the agents in both hardware and software mode. In previous phases the system required the agents to send different move command formats depending on if they were running on hardware or simulation. In addition, multiple software modules had to be started, and the set of modules required also depended on the mode. To achieve our goal of mode transparency we merged the simulator into the control server, which allowed the agents to interface with the control server in the same manner, regardless of mode.

In phase three the control server added additional movement types in hardware mode; agents now have access to these move commands in simulation mode. The control server receives the commands and then generates motor voltages if running in hardware mode, or velocities if running in simulation. Agents thus can use the same movement types regardless of mode, and have full functionality in simulation mode. To the agents the infrastructure looks and acts the same in hardware and software. This reduces the complexity of agent code by allowing the programmer to use one set of commands throughout development, rather than using one set to test in simulation, and a second set when validating in hardware. This reduces the design burden of new experiments, and helps the simulator more closely match real-life behavior.

The control server mode is set through a command line argument. Merging the simulator and the control server allows a user to run experiments with fewer modules, decreasing the system complexity and reducing the number of machines required to handle the infrastructure. It also allows the simulator to take advantage of any additional command types that might be introduced in the future, and reduces the number of places the code must be changed when new commands are added.

2.9 Conclusion

During the design we developed several design strategies in response to new requirements, or to overcome unforeseen difficulties. Often, problems required us to rethink our approach to the design, and redo parts of the test-bed. During this period we practiced some fundamental methods of design which helped reduce the amount of time spent rewriting portions of the code, or adding new modules. This section discusses some of the lessons learned, in an effort to assist anyone designing a similar test-bed.

2.9.1 Code Organization

In a large project, code organization becomes extremely important, especially as the number of people using or modifying the code increases. I recommend that a code control system such as CVS be used from the start of any major project, even if only one person expects to work on it. A code control system provides security from tampering, and a secure backup in case of error or catastrophe. It also helps maintain an organized structure to a project by requiring an active role by the user in structuring the project. A code control system is a vital part of maintaining code organization, especially with multiple users.

Having a modular code structure also greatly assists in keeping code organized, makes debugging code simpler, and makes code more easily reusable. In the test-bed, each task, such as checking tags, is separated into its own function, including any world state checks required before running the task. All operations related to the task are contained within the same function. This allows a user to look in one place to find all code related to a specific task, simplifying debugging and making the code more organized. This also makes it easier to use code in more than one module, as the user does not have to cut and paste from multiple functions. Obviously this principle can be taken too far, if every minor task gets its own function, but when used intelligently, it can significantly reduce the time required to finish a project.

2.9.2 Design

Often when starting a new project, it can be tempting to take existing software and modify it to the new purpose rather than design from scratch. Rather than take this approach, I recommend writing new software using relevant pieces from the existing software. This allows new pieces to be tested as they are added, without dealing with existing bugs or worrying about how the new code interacts with the old code. It also allows the designer to make design decisions without worrying about the ease of implementing them in the current code. While it may seem counter-intuitive, I have found that designing new modules from scratch and pasting in existing code when possible, leads to cleaner designs, and can actually require less effort.

For example, our original simulator was designed specifically for two teams of five robots with a static field setup. In addition, it required several threads to handle various incoming connections. For capture-the-flag we needed the simulator to allow for arbitrary numbers of robots on each team, and handle any valid maze setup. In addition we wanted a single threaded process. Rather than writing a new simulator using pieces of the old simulator, I modified the existing simulator, removing the threads and changing the way it handled robots and objects. Because the original was designed under significanyly different constraints, these changes required major code changes. The size of the existing code and its complexity made it difficult to determine whether bugs were the result of something I removed, or something I added. Because of this, making the modifications ended up being more difficult than originally estimated. In contrast, the the simulator in the final control server was designed from scratch to specific requirements. Useful pieces of the old simulator were cut-and-pasted as needed into the new simulator. This allowed us to test and debugg each new code section before adding the next. This process required less time and effort than making the original modifications, and resulted in software tailored to the application.

Chapter 3

Safe-path Graph Generation for Urban Environments

3.1 Introduction

Path planning plays a vital role in running experiments in a congested urban environment. While techniques exist that allow robots to navigate through such environments without planning an explicit path, such techniques cannot guarantee robots will reach their goal regardless of the maze structure. Effectively navigating a maze-like congested urban environment requires the use of a path planner. Path planners consist of two main parts: one that generates and maintains a graph that represents the world, and a second part that searches the graph to produce paths as needed.[8] This chapter focuses on the first of these two parts; many good solutions exist to search an existing graph, and any of these can be used as desired.

The importance of creating a good graph should be emphasized. The quality of paths created using the graph will be reduced if the graph is too sparse; if the graph is too dense, it will be expensive to search, increasing the run-time requirements of the path constructor. A good graph will provide something approaching the minimal representation of all safe movement options through the maze. Ideally, it will result in paths close to minimal length that avoid collisions with all obstacles. The safe-path algorithm was designed with all these conditions in mind, as this chapter will show.

The chapter is organized as follows: Section 3.2 discusses the lab setup and assumptions that inspired the creation of the safe-path algorithm. Section 3.3 describes the graph generation algorithms we initially implemented as we developed the game infrastructure. We explain the criteria we used to evaluate them and show how the schemes we implemented fell short in meeting our requirements. In Section 3.4, we present a new graph generation algorithm that avoids the problems encountered with the other approaches described here, and we show an example of a graph constructed using it. Section 3.5 presents data comparing the new scheme with two alternatives in terms of memory requirements and the quality of paths generated. Conclusions and future work are summarized in Section 5.2.

3.2 Lab Setup

As mentioned previously, our lab includes both hardware robots and fully compatible software simulators that can be used for a variety of applications. Figure 3.1 shows robots playing capture-the-flag. We currently have 10 omni-directional robots with on-board microprocessors that receive commands through a wireless modem. [9] The physical world of the robots is a square carpeted area that may be configured with wooden obstacles as desired. Each obstacle is 1 foot long, square in cross section, painted black, and fastened to the carpet with Velcro. Currently all obstacles are placed parallel to the X and Y axes of the field for simplicity's sake, but the algorithm works with obstacles of any polygonal shape and alignment.

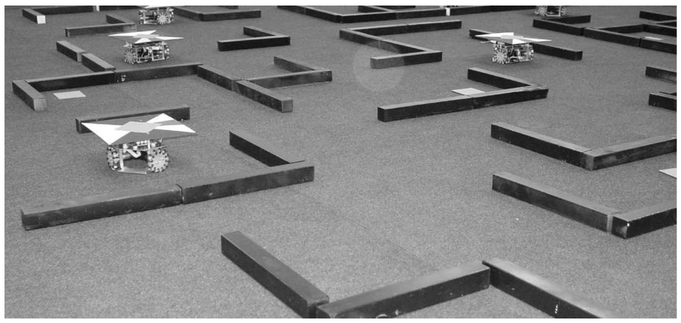


Figure 3.1: BYU MAGICC Lab Capture-the-flag Environment

Robots have on-board sonar but no vision capability; they are aware of their position and orientation. While we initially planned for robots to navigate through their world after receiving information from the UAVs about the layout of the obstacles on the field, the agents currently obtain map information prior to the game.

Future work will focus on requiring the robots to build map information through real-time exploration, in conjunction with the UAVs.

Our lab setup simulates real world multi-agent applications in which a UAV scouts out an area and transmits back to the robots the location and size of obstacles such as buildings and cars. Once they have a map, robots plan their own paths and navigate through the environment. In order to prevent collisions that could damage or disable robots, it is important that path planning algorithms produce paths that are *safe*, in that they maintain a minimum separation from adjacent obstacles.

3.3 Evaluating Path Planning Alternatives

Different environments present different path-planning challenges; this section analyzes the aspects of a graph that make it useful in a congested urban environment. During the early stages of the test-bed we implemented a Voronoi graph previously used for path-planning with aerial vehicles. One of the senior project teams also implemented a grid array scheme. Experimentation with both types led to the development of a set of requirements for a useful graph.

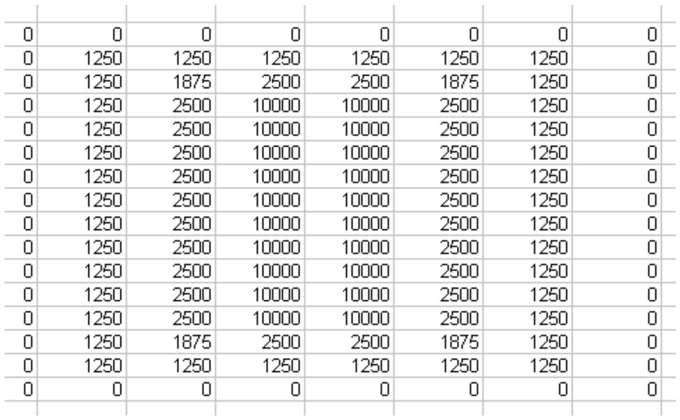
In order for a graph to work well in a path-planner for congested urban environment path-planner, it must satisfy four requirements. First and most importantly, a useful graph should account for obstacle dimensions in a way that creates safe "buffer zones" around all obstacles, but it must do this in a way that will not prevent access to any reachable areas of the world. Paths created from such a graph can therefore have any desired reachable point on the field as their destination, and they will always maintain a minimum distance from all obstacles. Such paths help mitigate the significant hazard to the robots of getting stuck on or crashing into an obstacle. Second, the algorithm should scale well with respect both to map size and the number of obstacles present. Our lab resources limit us to running the actual hardware robots in a relatively small area, but in simulation our experiments involve much larger maps producing much larger graphs, similar to real-world situations. Third, the graph should not require an excessive amount of memory to store. In a distributed environment, each agent must have its own copy of the graph, so total memory requirements are an important consideration. While we run nearly all the agent code on a PC, allowing us to ignore memory issues, most mobile robots have limited memory and computation capabilities. Future efforts to implement path-planning directly on the robots will require a graph with low memory usage. Finally, a useful graph will lend itself to implementation using a straightforward data structure that other modules in the agent code besides the path-planner can use if desired. Such a structure, when done well, also assists in fulfilling the other requirements.

An additional consideration when designing a graph depends on its usage. The design of a graph often makes obtaining minimal search speed and an optimal path mutually exclusive, requiring some sort of trade off. Because a graph returning optimal paths will likely be larger, containing more information, it tends to have increased search times, and vice versa. Real-time path planning requires minimal search times to maintain good response to user input, and allow proper control loop timing. In such cases the optimality of the path is of less concern than the time needed to find it. In the non-real-time case, optimality often is the primary concern, and one can safely ignore search time. Taking response time requirements into account during the design of a graph can make the difference between a useable graph, and a useless one.

3.3.1 Grid Arrays

The simplest type of graph to create is a grid array. [10] A gridded graph maps the obstacles in the world to an array of numbers. A large value representing infinite cost is placed in each element in the array that an obstacle maps to. The planner then creates a buffer zone of decreasing cost around each obstacle in order to create safe paths. The resulting array can be searched using any number of standard search algorithms. We used an $A*$ algorithm [11]. Figure 3.2 shows a possible representation of an obstacle in a grid array.

A grid array meets two of the requirements of a good graph. First, since it takes into account the obstacle dimensions when creating the graph, it can generate safe paths around the obstacles. However, getting a good margin of safety does



0	0	0	0	0	0	0	0
0	1250	1250	1250	1250	1250	1250	0
0	1250	1875	2500	2500	1875	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	2500	10000	10000	2500	1250	0
0	1250	1875	2500	2500	1875	1250	0
0	1250	1250	1250	1250	1250	1250	0
0	0	0	0	0	0	0	0

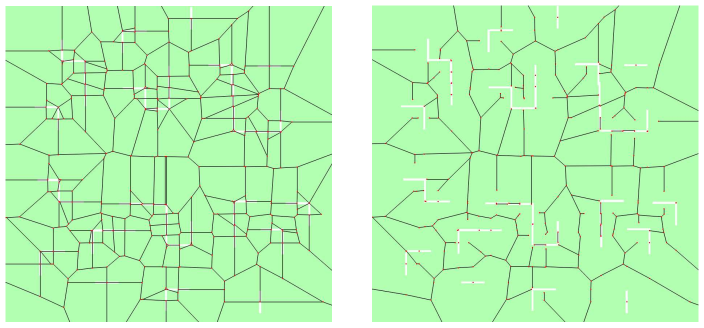
Figure 3.2: A representation of an obstacle in a grid array. The value 10000 represents the obstacle; lesser values indicate areas of relative safety near the obstacle. A value of zero indicates no danger of collision.

require some tweaking of parameters, particularly the amount of buffering, especially if the ratio of map units to array elements increases. A good buffering algorithm can alleviate this problem somewhat. Second, a grid array scales easily by increasing the array size, and increasing the number of obstacles does not affect the size of the array in any way. However, increased obstacle density can affect path quality, as the buffer zones of nearby obstacles merge. As the buffer zones merge, the path planner seeks paths with lower cost, bypassing the merged areas and creating longer paths. Increasing the granularity of the array can overcome this, but doing so exacerbates certain drawbacks of grid arrays.

Gridding fails to meet the third requirement because of the large amounts of memory needed to create the array. For example, the field our robots run on is a square, 4.5 m each side. Through experimentation we have found that we get the best paths when using a maximum ratio of 1 cm per array element. This ratio requires an array of 450 x 450 elements, each of which is a 4 byte integer, meaning the graph requires over 800 KB of memory to hold the array. The problem gets worse as the map gets bigger; doubling the length and width of the field requires 4 times the memory to represent. Reducing the size of the array by increasing the ratio of map units to array elements alleviates the memory problem somewhat, but also degrades the path quality, particularly in maps with large numbers of obstacles, again because of merging buffer zones. Finally, a grid array is not an ideal data structure for representing a graph because of its large size and because it cannot easily be accessed and manipulated by other graph-based algorithms without some additional structure. A grid array contains no information on nodes or edges, and thus cannot be easily used by some search algorithms, such as Dijkstra's.

The grid array exemplifies the trade-off between optimality and search speed. A grid array with high granularity returns very optimal paths, but the increase in size necessary to obtain such paths increases the search times significantly. Conversely, reducing the granularity to increase speed has a significant effect on path optimality.

3.3.2 Voronoi



(a) (b)

Figure 3.3: A Voronoi graph before (a) and after (b) pruning. Pruning causes some areas to become impassable by removing the edges through them, and does not remove edges that pass through small gaps between obstacles.

Voronoi graphs are often used for path planning, and numerous algorithms exist for generating one [12, 13, 14, 15]. The algorithm we implemented generates a standard Voronoi graph using the vertices of the obstacles as the starting nodes. After generating the graph, it prunes any edges that cross obstacles from the graph, leaving only valid paths. The size of the graph depends on the number of obstacles: increasing the area covered without increasing the number of obstacles does not affect the size of the graph at all. Increasing the number of points used to generate the graph, for example by including the midpoints of obstacle edges as well as the vertices, also increases the size of the graph, but provides better paths. In contrast to gridding, a large map can be searched as quickly as a small map, given similar numbers of obstacles, because they have comparable numbers of nodes and edges. Additionally, the graph has an organized structure consisting of a list of nodes and connecting edges which is a useful representation with a minimum of redundant information.

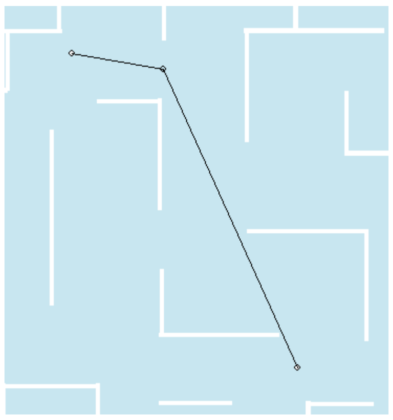


Figure 3.4: A path generated using a Voronoi graph. Notice how close the path comes to the obstacle. This could result in a collision.

Unfortunately, as Figures 3.3 and 3.4 illustrate, a Voronoi graph fails the first and most important requirement because paths created from it are more likely to bring robots within unsafe distances of obstacles. Voronoi graphs cannot explicitly account for obstacle dimensions; while it uses all vertices of an obstacle in its generation, the vertices have no connection to each other. When the obstacles are small relative to the robots this presents no problem, however in an urban environment this is not the case. Because of its inability to account for obstacle dimensions many of the created edges cross obstacles and should not be included in the graph. Before using the graph the path planner must prune the invalid edges. In our implementation, edges intersecting with obstacles were removed from the graph by checking each edge against the list of obstacles. Any edge intersecting an obstacle is removed. Adding more points to the initial set provides a better graph, but it also increases the amount of pruning required. Unfortunately, pruning can force the search algorithm to create suboptimal paths because some edges leading through open areas get removed from the graph. In extreme cases, some areas of the map become unreachable. As a final concern, unless complex pruning methods are employed, an unpruned edge can lie arbitrarily close to the edge of an obstacle, leading to unsafe paths.

After determining that neither of these two graph generation approaches fully satisfies our requirements, we realized the advantages of an approach that uses obstacle geometry in generating the graph. After some experimentation and iteration, we developed the algorithm described in the next section. This algorithm offers several advantages: paths generated using the graph it generates will provide safe paths to all accessible map areas, it scales well with map size and obstacle density, it has relatively low memory requirements, and it is represented using a logical data structure allowing access to all nodes and edges in the graph.

3.4 A New safe-path Graph Generation Algorithm

Our algorithm begins with a list of obstacles, the desired buffer distance around each obstacle, and the maximum map dimensions. Each polygonal obstacle is represented by a list of points that make up the vertices, in order around the edge of the polygon. Using these values we can create the line segments that make up the edges of the obstacle. The buffer distance is the minimum safe distance a robot can get to an obstacle — generally slightly more than the maximum radius of the robot. The user can vary this value as necessary. For example, a sensor-based obstacle avoidance algorithm can reduce the buffer distance needed to create safe paths. The initial version of the algorithm assumed that all obstacles were rectangular and aligned with one of the map axes, but after some modification the algorithm can now be applied to maps with convex polygonal obstacles of any orientation. Concave polygons need to be subdivided into their component convex polygons - a task for which algorithms already exist [16].

Generating our graph consists of five main steps, one of which is an optional optimization. In addition we have developed a line smoothing algorithm that can be applied as the graph is used for path planning.

3.4.1 Create Buffer Zone

First, we expand each obstacle in the list by the buffer distance. In doing so it is important that the exact obstacle dimensions are maintained, so that the expanded polygon is directly similar[1] to the original. Each line segment of the expanded polygon will be parallel to exactly one line segment in the original, with the buffer distance as the perpendicular distance between them. Expanding the obstacles makes each line segment in the expanded obstacle's border the closest path a robot can follow without hitting the obstacle.

To ensure the expanded obstacle maintains direct similarity to the original, any concave polygons must first be broken into convex parts. The expansion algorithm does not guarantee direct similarity for concave polygons, and will produce erroneous results. Fortunately, it is a fairly simple matter to create convex polygons from a concave polygon [16].

Having obtained a set of convex obstacles, we run algorithm 3.1 on each. We calculate the geometric center of the polygon using an algorithm described by O'Rourke [16]. The polygon is then translated so the geometric center becomes the origin. One of the difficulties in expanding a non-rectangular polygonal obstacle is ensuring the line segment is expanded in the proper direction, ie. we don't inadvertently shrink or deform the obstacle by expanding line segments in the wrong direction. By stipulating the use of convex polygons, and setting the origin at the geometric center, we ensure that any line segment with a positive y-intercept will only need to

[1]all corresponding angles are equal and described in the same rotational sense

be expanded in the positive y direction, and vice-versa for those with negative y-intercepts. In addition we ensure that no part of the hull will pass through the origin, which simplifies certain steps. These assumptions hold for vertical line segments; the x-intercept simply replaces the y-intercept. The expansion algorithm (3.1) can start with any vertex, and proceeds through the remaining vertices in clockwise rotation around the hull of the polygon.

Note that the values of the starting line, m_i and b_i, are saved for use in the final step. In addition, the algorithm saves the slope and intercept calculated in the previous step for use in the current step. Thus, each value is calculated only once during the algorithm, increasing its speed. Observant readers will point out that θ can actually appear in either of two quadrants, depending on how it is calculated. This makes no difference, as only the sign will change, not the value, which is why the absolute value of r is used. Finally, while vertical lines complicate things somewhat, the essential functioning of the algorithm remains the same; the equations are just rearranged somewhat.

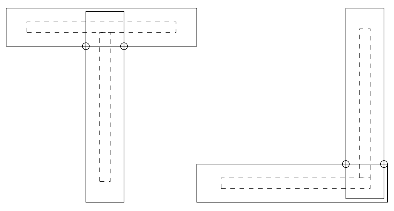


Figure 3.5: Dotted lines outline the actual obstacles. Solid lines show the "buffer zone" created around each obstacle. The circles denote where the line segments will be cut.

Algorithm 3.1 Expand Obstacles

```
1: for Each polygon with $n$ vertices do
2:   Calculate the geometric center of the polygon. Translate vertices to set as the origin.
3:   for $i = 0, i \to n$ do
4:     Find the line segment $l_i$ formed from vertices $p_i$ and $p_{i+1}$
5:     if $l_i$ is non-vertical then
6:       For $l_i$ calculate the slope $m_i$, and y-intercept $b_i$
7:     else
8:       For $l_i$ calculate the slope $m_i$, and x-intercept $b_i$
9:     end if
10:    Find $\theta_i$, the angle of $l_i$ with respect to the origin.
11:    $r = expandamount / \sin(\pi/2 - \theta_i)$
12:    if $b_i > 0$ then
13:      $b_i = b_i + |r|$
14:    else if $b_i < 0$ then
15:      $b_i = b_i - |r|$
16:    end if
17:    Save the values of $b_i$ and $m_i$ for later use
18:    if $i! = 0$ then
19:      Set the intersection of lines $l_i$ and $l_{i-1}$ as a vertex in the expanded polygon.
20:    end if
21:  end for
22: end for{The final vertex is formed from the intersection of lines $l_0$ and $l_{n-1}$}
```

3.4.2 Remove Invalid Segments

After expanding the set of obstacles we create a list of line segments from the contiguous vertices of the polygonal hull[2] of each obstacle. As stated before, the obstacles can overlap; this step removes line segments interior to the outer hull of overlapping obstacles. The process is described in algorithm 3.2.

Algorithm 3.2 Remove Invalid Segments

```
 1: for Every obstacle, $o_i$ do
 2:     for Every obstacle $o_k, k \neq i$ do
 3:         if Bounding circles intersect then
 4:             for Every edge $e_{i,m}$ in $o_i$ do
 5:                 if Both endpoints of $e_{i,m}$ are inside $o_k$ then
 6:                     Remove $e_{i,m}$ from edge list
 7:                 else if One endpoint is inside $o_k$ then
 8:                     Find intersection $cept$, of $e_{i,m}$ with hull of $o_k$
 9:                     Set the endpoint inside of $o_k$ to $cept$
10:                 end if
11:             end for
12:         end if
13:     end for
14: end for
```

The list created in the process described above may still contain some edges that cross through obstacles, and thus must be removed. This is because while algorithm 3.2 removes all segments with endpoints that lie inside another obstacle,

[2]The convex hull of a polygon P is the smallest-area convex polygon which encloses P. The convex hull of a convex polygon P is P itself

some edges crossing an obstacle will have both endpoints outside the obstacle and will not be removed. To deal with this, we find all intersections of the line segments in the list, as shown in Figure 3.5. This step requires the most time in the graph generation algorithm, as each line segment must be checked against every segment. Where a line segment intersects another at a non-endpoint, we add the two smaller segments making up the bisected line segment to the list and remove the original (larger) segment. This guarantees the list contains segments of minimal length and that segments intersect only at their endpoints. It also guarantees that no segment crosses any obstacle without having an endpoint contiguous with the obstacle hull, and allows us to remove invalid segments without breaking the path around the obstacle. We achieve this in much simpler fashion than algorithm 3.2, because we only need remove those segments whose midpoint lies within an obstacle and not contiguous with its hull. We can make this simplification because our list of segments contains only segments minimal in length, with no intersections other than at the endpoints, and whose endpoints lie contiguous with some obstacle's hull. After removing all such invalid segments we find and delete any duplicate segments, in order to save computation time in the following steps. The resulting set of line segments includes all minimally safe paths around the obstacles and guarantees that a robot following any segment will not hit an obstacle, as seen in Figure 3.6.

3.4.3 Connect Sections

The line segments created in the previous steps do not yet constitute a useable graph so we must add additional line segments to connect the separated sections. There are several ways to do this, depending on the graph characteristics and the speed of generation desired. We have tried three alternate methods. The first method works best when all obstacles are rectangular and aligned with one of the grid axes, and is the method we use for the mazes used in the MAGICC lab hardware setup. We create two lines, one parallel to each axis, that extend from each corner of every obstacle out until they intersect another expanded obstacle, or the map edge. This method creates a well-connected graph with relatively low computational overhead.

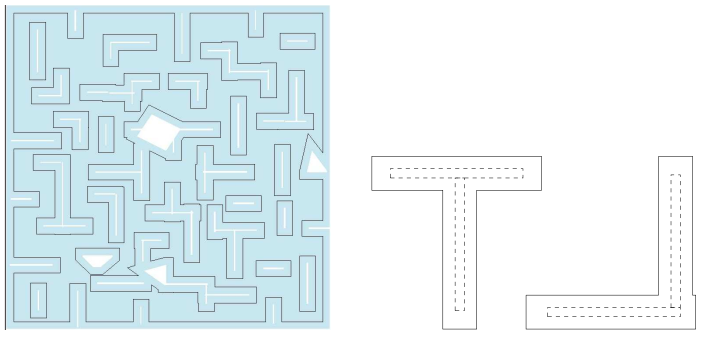


Figure 3.6: (a) The graph with polygonal obstacles after removing invalid line segments. The actual obstacles are shown in white. (b) A standard map after removing invalid line segments.

Figure 3.7 shows this method. The second method involves finding every possible connection from every vertex of every obstacle, and adding those connections that don't intersect any other obstacle. This method creates the most complete graph, but is extremely slow. The third method only looks for the n closest connections to each vertex and connects them. This significantly reduces the computational requirements, but results in a less complete graph than method 2. However, as will be shown in a subsequent section, this is not a significant disadvantage, and the increase in speed more than makes up for the reduced number of connections. There are many other possibilities that could be explored.

3.4.4 Optimization

At this point we have the option of performing an additional step to create a smaller, cleaner graph. Finding the intersections between the connecting lines and breaking the lines into the component segments as we did in step two will increase the number of line segment endpoints, and help us combine nodes in the final step. Depending on the connection method being used, it can also create a better initial

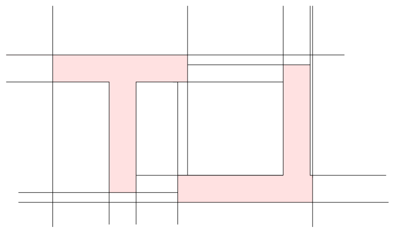


Figure 3.7: Connecting the sections. Line segments are drawn from each corner to the nearest obstacle. Some parallel lines will be removed during merging. The grey sections denote the original expanded obstacles.

graph by allowing more lines to be merged. During this optimization step we also remove any duplicate line segments created in the previous step, reducing the computation time required to create the final graph. Finding all the intersections can require a substantial amount of computation time on maps with large numbers of obstacles. In practice we have found that omitting this step does not create any noticeable problems with the final graph. Because the computation time required to perform this step is usually greater than that of any other step, omitting it speeds up the overall time substantially. A graph created without this step will be 5-10% larger than one created using it, because there will be fewer nodes available for merging in the final step. One should note that in a sparse map, if one uses the first method of graph connection described in the previous section, omitting this step may result in parts of the graph being unconnected if the lines created in the previous step for an obstacle do not intersect with any other obstacle. This is because of the way the final step generates the graph from the line segments, and can be avoided by using either of the other two methods described above. This step provides greater benefit on sparse maps; its value rapidly diminishes as the map density increases.

3.4.5 Create Nodes and Edges

The final step takes the list of line segments and creates the graph from it. The finished graph consists of a list of nodes, each containing a position value and a list of connections to other nodes. To create it, the generator takes each line segment

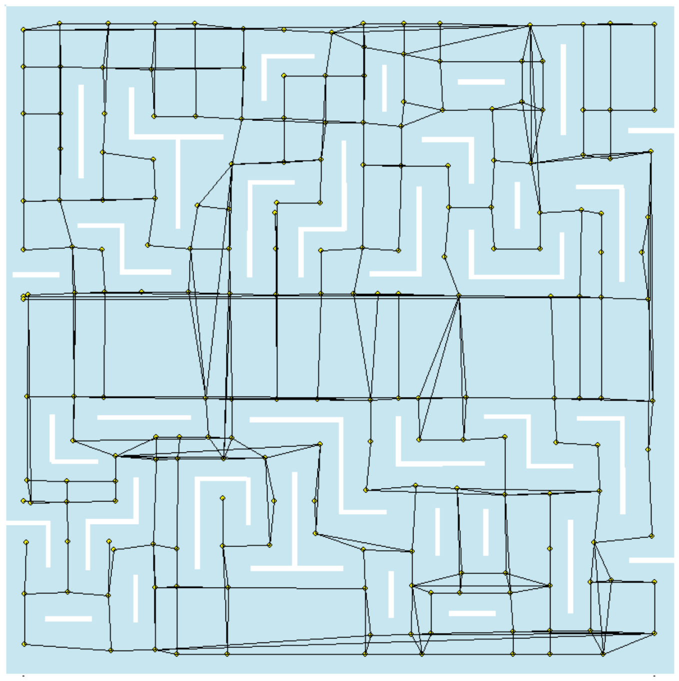


Figure 3.8: The final graph.

in the list and finds its endpoints. It then checks the graph to see if either endpoint has previously been added to the graph; if not, the generator creates a new node for each unique endpoint, adds the connection to both nodes' list, and then adds the new nodes to the graph. If the graph already contains a node for one or both endpoints, the graph adds the connection to the list of the existing node(s), but does not add a new node to the graph for the duplicate endpoint.

Once the generator has added all the line segments, we can clean up the graph by merging nodes within a certain distance of each other into one node. The generator merges nodes by taking two close nodes, setting the position of one node to the average of the two nodes' position values, unifying the set of their connections, and then removing the second node from the graph. One should note that merging does not remove any edges from the graph, only nodes. Merging reduces the number of nodes in the graph, reducing the graph's size, but merging can bring nodes and connections closer to obstacles by averaging the positions, reducing the safety margin. The user can define a distance between zero (no merging) and infinity (all nodes merged into one) for the merges. An optimal value for the merge distance will increase the quality of the graph significantly by reducing the number of redundant and useless nodes without reducing the available connections. We have found that a value approximately half the distance of expansion provides optimal merging of redundant nodes without significantly reducing the margin of safety.

3.4.6 Path Smoothing

After merging, the graph can be used with the search algorithm of choice, however we have added one final optimization. This optimization actually occurs during path generation by dynamically adding connections to the graph. Because of the extreme computational requirements of creating a complete graph (one with all possible connections) most graphs created will be suboptimal. As a result any path created using that graph will also be suboptimal. This results in unnecessarily 'jagged' paths, which slow the robot down by causing it to repeatedly change direction. The path smoothing algorithm reduces the effects of using a suboptimal graph by combining the jagged path segments when possible, and adding new connections to the graph as it finds them. After the path planner generates a path from the graph the path smoother checks the path and removes unnecessary points, shortening the path and straightening out some path segments.

The path smoother keeps track of three points — the current point, i, the last valid point, j, and the point to check against, k. To begin, i is set to the first point in the path, j is set to the second point in the path, and k is set to the third point. The line segment between the points i and k is then checked to see if it intersects with an edge of any of the expanded obstacles. We use the expanded obstacles to ensure the smoothed path maintains the desired safety margin. If the path segment does not intersect any edge, then a robot can travel directly from point i to k without going through j, and j can be deleted from the path. If segment (i, k) does intersect then the robot cannot bypass point j without hitting an obstacle and we must keep j in the path. The current point i is set to the last valid point j, and j is set to k. We then increment k and repeat the process of removing unneeded points until we reach the end of the path.

Algorithm 3.3 shows the process in shorter form. Note that in the code implementation we do not actually remove the points from the path, instead we create a new vector object containing only the valid points. Creating a new vector instead of deleting elements from the old one actually speeds up the algorithm, and prevents pointer errors in the old vector caused by deleting elements while still iterating through the vector.

Algorithm 3.3 Path Smoothing

```
1: Set $i = 0, j = 1, k = 2, n = path.size()$
2: while $k < n$ do
3:     while line $l_{p_i,p_k}$ does not intersect any extended obstacle AND $k < n$ do
4:         Remove $p_j$ from path
5:         $p_j = p_k$
6:         $p_k = p_{k+1}$
7:     end while
8:     $p_i = p_j$
9:     $p_j = p_k$
10:    $p_k = p_{k+1}$
11: end while
```

After smoothing the path we add any new connection to the graph created by the points $(i,\, i+1)$ in the path. We do this after smoothing is complete to reduce the number of new connections added, and ensure that only those connections actually used get added to the graph. As the planner creates paths the line smoother will continually add additional connections to the graph. The new connections increase the size of the graph somewhat, but they also speed up the line smoother by reducing the number of points that must be removed from the path, because the planner will use the new optimal connections when creating the path. The number of new connections added will also decrease with time, helping limit the growth of the graph.

It may seem like the line smoother would continue increasing the size of the graph arbitrarily, however as more connections are added, the planner uses them to create better paths, and fewer new connections need to be added. If the additional graph size and computation time involved in using the line smoother proves prohibitive, it can be turned off without rendering the graph unusable. As noted before, when not using path smoothing, the path planner will return longer paths with more points than those created with line smoothing on. Unsmoothed paths will also tend to zigzag more than smoothed paths, and can seriously degrade the average speed of a robot following the path. In practice we have found the line smoother generally requires only slightly more time to complete than the search algorithm, and that the benefits, both in better paths and an improved graph outweigh the additional computation time required.

3.5 Results and Analysis

To see how our graph performed against gridding and Voronoi we ran several tests. In addition, we have tested the graph in dozens of hours of actual game-play. For the first test we planned a large number of paths using each type of graph. We planned the paths using the same set of random starting and ending locations for all graph types to ensure a fair comparison. To reduce the probability of a single map affecting the performance of any particular graph type, we planned 160 paths on each of three different maps, for a total of 480. After generating each path we computed the minimum distance that path came within any obstacle, and computed the average minimum distance for each of the three graph types. This average gives an approximation of the relative safeness of paths generated using each type of graph. Table 3.1 shows the results. When created with a buffer zone of 120 mm, our algorithm generated paths with an average minimum distance 22% greater than the grid array,

Table 3.1: Average minimum distance from an obstacle for paths generated from each type of graph. Values given in mm.

Minimum Dist. to Obstacle	
safe-path:	112
Grid Array:	92
Voronoi:	54

and 107% greater than Voronoi. The fact that the minimum safe distance was less than the specified 120 mm can be attributed to the node merging algorithm, which as we noted before causes paths to move closer to obstacles. Omitting the node merging step brings the minimum distance to 120 mm. We can tweak the grid array's buffering algorithm to achieve approximately the desired distance of 120mm, but every time the desired value changes it requires recalibration. Safe-path creates the desired distance during generation without requiring any changes to the algorithm.

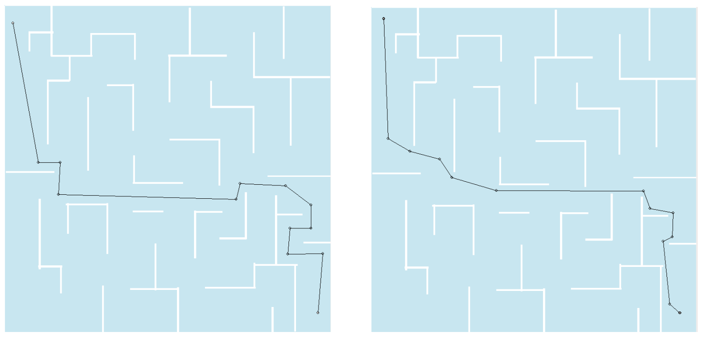
After running the first tests, we calculated the memory requirements of each graph, not including any additional memory used while searching the graph. Our Voronoi implementation and graph had very similar memory requirements - 14 KB, and 10 KB respectively for our field with 50 obstacles. As noted previously, gridding requires 800 KB for the same map with a ratio of 1 cm per element. With a simulated map $100m^2$ containing approximately 250 obstacles, Voronoi required 80.6 KB, graph required 34.4 KB, and gridding required 3.2 MB for best results. As can be seen, both Voronoi and graph scale fairly linearly with the number of obstacles. Gridding scales well with the number of obstacles, but poorly with map size because of the extra memory required for larger arrays; doubling the map dimensions quadruples the memory required.

Comparisons of graphs for the final requirement are somewhat subjective, but safe-path and Voronoi do well, since both types consist of a list of nodes and edges with little redundant or useless information, while a grid array contains large amounts of unnecessary information, and no correlation between elements.

One final comparison is that of generation time. While low generation overhead was not one of our main requirements, an algorithm with excessive generation overhead will not be useful in real-time conditions. The overhead to generate a graph using the safe-path algorithm is equivalent to that of gridding or Voronoi for maps with less than 50 obstacles. However, it increases as more obstacles are added, and with more than 100 obstacles the generation overhead becomes significant. In a large map with 250 obstacles, generation using the safe-path algorithm required between 18 and 20 seconds on a Pentium 4 2.5 HZ computer, compared to 1.5 - 3 seconds to generate a grid array or Voronoi graph on the same machine. Reducing the length of the generation is suggested as a major focus of future work on the algorithm. We note that the overhead for generation is paid just once using safe-path or gridding, so computational efficiency is not a major concern. It should be noted also that while a safe-path or grid array graph remains static once generated, a Voronoi graph requires regeneration every time the path changes. The algorithm must use the starting and end point of the path during generation, and because any change in one part of a Voronoi graph can potentially affect the whole graph, it must regenerate the graph for every path. More advanced versions of the Voronoi graph may not suffer from this defect, but in the simple case the Voronoi generation overhead must be paid every time a path is generated.

We measured the execution time for planning paths using the three graph types, but since the different graph types require different search algorithms, the comparisons are not meaningful. While our approach and Voronoi produce similar data structures and can therefore be searched with similar algorithms, the grid array requires a different approach with substantially more overhead. We note that, using Dijkstra's search [17] on a moderately sized-graph of approximately 40 obstacles, our path planner generally required less than 2-3 ms to generate a path when using path smoothing, and less than 1 ms with no smoothing. We have run several hours of testing for the three types in actual game situations, with the following conclusions:

The grid array produces the shortest paths, with reasonable safety margins, but suffers from long search times. The long search time causes significant response delay to user input; in addition the agent AI routines take too long to respond to changing game situations, causing poor performance. In a less time-critical application gridding would have great application. The Voronoi approach works best when used in conjunction with some sort of obstacle avoidance routine. This helps minimize the effect of the low safety margin its paths provide. However, the tendency of Voronoi to create graphs with no access to certain areas makes it completely unusable on certain maps, and unreliable in general. The safe-path algorithm provides a good margin of safety, and while not providing as optimal paths as gridding, the fast search times provide the best response of any of the three types. Safe-path has proved its utility over dozens of hours of use, with several different map types and multiple interface types.



(a) (b)

Figure 3.9: (a) A path generated using the safe-path algorithm. (b) A path generated using a grid array.

3.5.1 Hardware Results

This algorithm was developed and tested using our simulation framework. We have also tested the algorithm extensively using hardware robots. These hardware tests confirmed that the safe-path graph's ability to maintain a higher minimum distance from obstacles creates an advantage for robots by reducing the number of collisions with obstacles, especially when used in conjunction with an obstacle avoidance algorithm. This ability to reduce collisions reduces the chances that a robot will become non-functional due to a collision and reduces the amount of time a user is inconvenienced by robots that have become stuck on obstacles. It also reduces wear and tear on the robots. While we have not performed nearly as many hours of hardware tests as in simulation, the nature of the test-bed ensures that safe-path will function similarly in hardware as in simulation, where it has proven utility.

Chapter 4

Multi-Agent Experiments on the MAGICC Test-bed

4.1 Introduction

While the development of the test-bed was a learning experience, it is not an end in itself. This chapter discusses several of the experiments run on the test-bed by other researchers, and the functions of the test-bed that made them possible. It does not present conclusions reached, or data gathered by the researchers, as that is beyond the scope of this thesis. Interested parties can consult publications by other researchers for more information.

4.2 Standard Experimental Setup

This section describes the basic setup of the environment for directing a team of robots. It details the interface, structure, and capabilities of the basestation and robots, and gives examples of its use.

In order to more easily develop and run experiments, I have created a basic interface that includes a basestation GUI, some robot controllers and the basic communications infrastructure in the robot agent. This should allow future users to quickly add appropriate agent code, without the need for agent infrastructure development.

4.2.1 basestation

The basestation was developed using QT, and provides a graphical interface for interacting with all the agents on a team. Figure 4.1 shows the display of the basestation. The interface is divided into two parts: the map, and the toolbar. The map displays the obstacles, the agents on the team, and the locations of any enemy

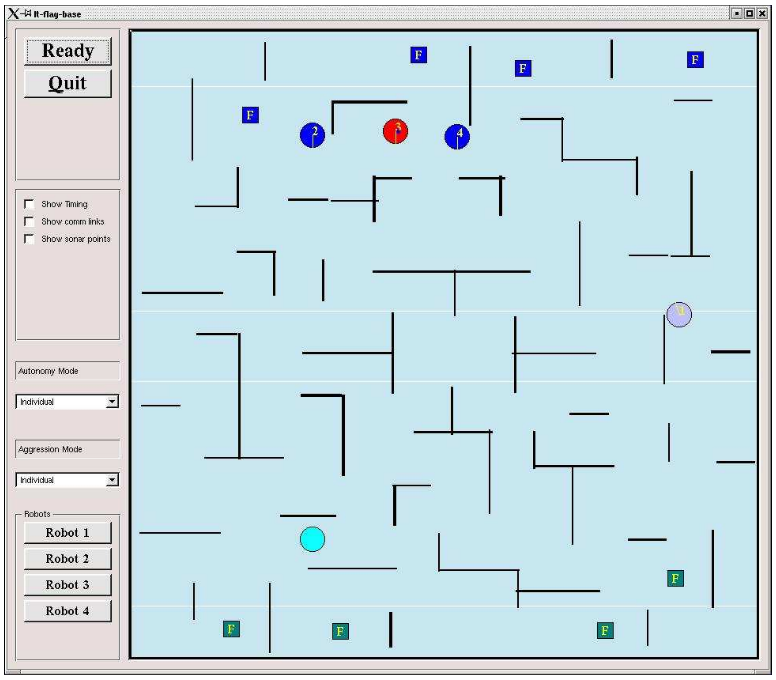


Figure 4.1: The basestation GUI. Friendly robots, flags and UAVs are blue, while enemies are green.

robots that have been detected. The basestation uses the configuration file to generate the obstacles displayed on the map. Clicking on a robot selects that robot and allows the user to issue commands to the robot. Multiple robots can be selected and issued commands by holding down the control key during selection. The checkboxes on the left allow the user to toggle output of timing information to the console, display of the communication radius of each agent, including the basestation, and display of the robot's sonar information.

Each robot has a mini display window that can be brought up by pressing the corresponding button in the lower left part of the GUI. The mini display window holds tabs for movement, commands, and status. The movement tab allows the user to select the path-planning type, and manually direct the robot using a 'driving' widget. The command tab presents buttons for tag, win, surrender, pick up and drop flag, and attack and defend flags. The attack flags button sends the robot to attempt to pick up each enemy flag, while defend flag creates a cyclical path between all the friendly flags, which the robot follows. If it senses any enemy robots with its sonar it will chase them and attempt to tag them. If successful, or if it loses track of the enemy robot, it returns to its old path.

4.2.2 Robot Agent Structure

The robot agents' structure consists of some initialization code, followed by a while loop containing the behavioral code. After each iteration of the loop the agent pauses, releasing the CPU so that other processes can run. The loop can run anywhere from 1-100 iterations per second. The agent process is timing-sensitive only to user commands, meaning that there must be no noticeable delay between the user giving a command, and the agent's acknowledgement. The behaviors themselves are timing-insensitive, meaning that delays between loop iterations will generally not cause erratic behavior. The exceptions to this are reactive behaviors such as chase, evade and tag. Any delay must be short enough that the situation does not change dramatically between iterations. Practically speaking this is not a problem, as the delay is generally only a fraction of a second between iterations. In practice, the loop timing is set at about 10 iterations per second. This provides acceptable performance with teams of 4 or more agents, and still reacts well to user input and changes to the world state.

User commands are sent to the individual robots in string format, allowing for easy debugging. Table 4.1 shows the list of commands currently accepted by the robot agents. The agents parse the command and then call appropriate functions to handle them. At each time-step agents send their position data to all other agents within range. The UAV sends all other agents, including the basestation, the location of flags as it finds them; subsequently it transmits only changes in flag location. The UAV also sends the locations of any enemy robots it sees. Robot agents use this world state information to make decisions, or as part of individual behaviors. For example, the AutoPickup behavior checks the position of enemy flags vs. its own position, and when properly positioned, issues a pickup request to the Referee.

Table 4.1: Robot Commands

Command	Issuer	Function
AgressionMode	basestation	Sets the aggression mode
AutonomyMode	basestation	Sets the autonomy mode
CyclicalWaypoint	basestation	Adds cyclical waypoint to the agent's list
DefendFlags	basestation	Triggers DefendFlags behavior
DropFlag	basestation	Tells agent to issue drop flag request
GenerateCyclicalPath	basestation	Generates cyclical path using point list
GetFlag	basestation	Tells agent to issue a pick up flag request
GoToPoint	basestation	Agent plans a path to the specified point
PathType	basestation	Sets the path planner to use the specified type
Quit	Referee	Causes the program to exit
Restart	Referee	Alerts agent that the game has restarted
SetVel	basestation	Sets robot's velocity to the specified value
Start	Referee	Alerts agent that the game has started
Stop	Referee	Alerts agent that the game has stopped
Surrender	basestation	Tells agent to issue surrender request
Tag	basestation	Tells agent to issue tag request
ToggleChasing	basestation	Toggles chase mode
ToggleEvading	basestation	Toggles evade mode
ToggleSonar	basestation	Toggles sending of sonar data
TurnToPoint	basestation	Robot turns to face the specified point
Untag	basestation	Tells agent to issue untag request
Waypoint	basestation	Extends current path to go to the specified point
Win	basestation	Tells agent to issue win request

4.3 Autonomy and Aggression Levels

Interfaces allowing one human to control multiple robots engaged in multiple separate tasks while minimizing the total idle time for the robots present numerous areas of research. Using the MAGICC test-bed, I expanded the basic basestation and robot agents described above, using the concept of aggression levels coupled with autonomy levels. Aggression levels moderate the behavior of a robot, while the autonomy level moderates the amount of decision making an agent will do on its own. Coupling these two allows a user a lot of flexibility in setting the behavior of the robots.

For this project I wrote a large number of self-contained low-level behaviors that can be used to build higher-level behavior. For example, the AutoEvade behavior causes the robot to move away from enemy robots when it senses them with its sonar. The AutoWin behavior plans a path back to the robot's home side whenever the robot picks up the enemy real flag. When combined, the robot will evade enemy robots while attempting to move back to its own side. Without the evade behavior the robot will move directly back to its own side, ignoring any enemies sensed along the way. Together they work to return the robot safely home with the flag. Many of these behaviors can be explicitly accessed using the setup described in Section 4.2; what I have done here is combine them in useful ways, allowing the user to select the types of behavior a robot should exhibit. Table 4.2 provides a list of the autonomous behaviors I developed and a short description.

The autonomy modes define how much work the agent will do on its own. An agent with no autonomy will not do anything without express user input. The user must tell it to pick up flags, tag enemies, where to move, etc. This mode provides the basis for measuring the efficiency of the other modes.

A partially autonomous agent will tag enemies that come within range, pick up flags when moved to the proper position, move to the untag zone when tagged, and in general do most low-level behaviors on its own as the situation warrants. It does not make any decisions on its own, however, and will not coordinate its actions with other robots unless specifically directed by the user. This mode frees the user from

Table 4.2: Autonomous behaviors

Behavior	Description
AttackFlags	Visits each enemy flag and attempts to pick it up
AutoChase	Chases enemies detected by sonar
AutoEvade	Evades enemies detected by sonar
AutoPickup	Picks up flag when in suitable position
AutoTag	Tags enemies who come within tag range
AutoUntag	Plans path to untag zone; requests untag upon arrival
AutoWin	Plans path home after picking up flag; requests win upon arrival
DefendFlags	Generates cyclical waypoint between flags

having to manually execute basic behaviors, while still making strategic decisions. For example, the user would still have to direct the robot to the proper position to pick up an enemy flag. Once there, the agent would automatically attempt to pick up the flag and return home if successful. Robots in this mode can be set to go to full autonomy mode after some time delay with no user input received. This reduces the effects of neglect when a user has more robots than they can handle efficiently. The agent returns to partial-autonomy mode after receiving any input from the user.

Fully autonomous agents make all strategic decisions for themselves, based on the current game state. They base their decisions on the current aggression mode selected by the user, the location of other robots (friendly and unfriendly) and flags. The user can override the current behavior selected by the agent, but the agent will resume its activities after following the user directive.

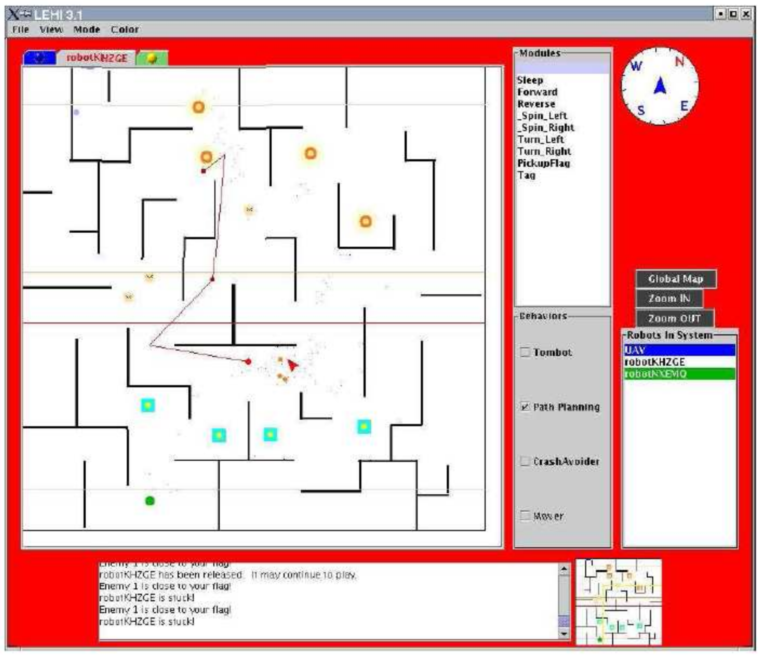
4.4 Interface Studies

The test bed has shown its usefulness in a study by the Human-Centered Machine Intelligence Lab (HCMI Lab) associated with the Computer Science department at Brigham Young University. This study, done by Josh Johansen, focused on the effects of interface design and functionality on neglect tolerance, situation awareness and attention management. For this study they developed the GUI shown in Figure 4.2. This GUI acts as the basestation, and also contains the agent code for all robots on the team. Each robot has a separate map display which the user switches between using tabs on the GUI. Users playing the capture-the-flag game randomly had their attention removed from the game by the GUI, which required them to answer math questions before they could resume play. While they answered the questions the game went on, requiring the user to adapt to a changed situation on return. The game was played with various assistance options turned on or off, and the GUI measured the performance of the user with each option for comparison purposes. The GUI provides four options to help the user: a fading tail, a waypoint manager, sonar memory, and an attention manager.

The fading tail provides the user short-term, visual feedback of a robot's path, by providing a 'tail' following the robot. The end of the tail fades out over time, to prevent the screen becoming overloaded with information. The tail should help a user rapidly assess what the robot has been doing while his attention was elsewhere, and backtrack if necessary.

The waypoint manager allows the user to set points that he would like the robot to go to. The agent plans its own path between the points, freeing the user from unnecessary micro-management of the robot's movement. The user also has the option of modifying individual points along the path, or of making the whole path cyclical. The waypoint manager allows the user to set the robot's behavior for a longer period of time, reducing the effects of neglect.

In the HCMI lab GUI the map is not processed and displayed before the game begins, so they have developed a method for the robots to use their sonar to discover the layout of the map. The sonar memory filters out motionless objects such as obstacles and displays them on the GUI, slowly building up a map as the robots move about the field. In addition, mobile objects are compared against the information collected, allowing the user to identify enemy robots. The sonar memory



```
Enemy 1 is close to your flag!
robotKHZGE has been released.  It may continue to play.
Enemy 1 is close to your flag!
robotKHZGE is stuck!
Enemy 1 is close to your flag!
robotKHZGE is stuck!
```

Figure 4.2: The GUI used in the human-factors work done by the Human-Centered Machine Intelligence Lab at BYU.

helps the user understand the robot's position relative to the objects on the field and react to enemies.

The attention manager provides visual or audio feedback when a robot requires attention, e.g., a robot cannot complete a command, becomes stuck or tagged, or has some other unexpected event requiring user input. The attention manager uses different signals depending on the urgency of the problem. For example, if a robot is unable to complete a command, the attention manager signals the user through beeps, flashes and text, while lesser problems get communicated with simple oscillating colors. The attention manager allows the user to divert his attention for a longer time—span, and still respond to problems with the robots. It also helps the user respond to problems in an efficient fashion, by signalling which problems are most urgent.

4.5 Formation-based Team Control

Jeff Anderson's thesis work in multi-agent studies involves the use of formations to control teams of robots in the completion of a task. Based on previous work by Tucker Balch and Ronald Arkin [18], Jeff uses formations directly with other behaviors to assist human operators in controlling a team of robots. After setting the formation, the user can then control the robots as a group, commanding them to move to a location, or follow enemy robots for example, as a single entity. He has shown that the robots can maintain a formation while executing other behaviors. The formations act as building blocks for more complex team behaviors.

In addition to research on formation control, Jeff is running experiments on neglect tolerance and human factors using the formations and team behaviors already developed. Neglect tolerance is a metric used to analyze an agent's performance when neglected by the human operator. A neglect-tolerant agent will have a smaller degradation than one with lower neglect-tolerance. Neglect is simulated by interrupting the user's control of the team by suspending the display of the field temporarily. The robots must run on their own for a certain period of time before the user is allowed to resume control. The efficacy of each control scheme is then measured by the time it takes to accomplish the task while suffering the forced interruptions. The study hopes to show that using team-based formation behaviors provides more neglect-tolerance than having the user control individual robots.

Chapter 5

Conclusion and Future Work

5.1 Multi-agent Test-bed

During our work on the test-bed we struggled to balance two important principles. First, overloading any single critical process in the system will slow the entire system down; the balance of functions between system processes plays a critical role in determining system performance. Second, too many processes make the system unwieldy and difficult to use. A large number of critical processes creates substantial system complexity in the code organization and structure, the communication system between processes, and in understanding the relationships between modules. In addition to being unwieldy for use in experiments, such a system increases the difficulty of maintaining and updating the code, increasing the likelihood of its obsolescence. Finding a good balance between these two competing principles early in the system's design will reduce the amount of time spent later in system revisions. In our design we went from one extreme to the other, with less than satisfactory results. Our final design achieved a good balance in the number of processes and the functions they perform, with little redundancy. Each process performs a minimal set of related functions, without doing too much or too little.

The design of our timing system for the loops played an important part of our ability to achieve this balance. In the early designs the timing loops used the CPU in an inefficient manner, leading us to believe we had less processing power available. This in turn caused us to increase the number of processes, leading to a larger more complex system in an attempt to reduce the amount of work allocated to each process. With a better timing strategy we were able to increase the workload given to each process, substantially reducing the size of the system. Properly estimating the capabilities of the hardware the system runs on would have alerted us sooner to the real source of our performance problems.

Having overcome our original errors, we found it useful to place similar functions into a single process. For example, placing the hardware control loops and simulator into the same process allowed for complete mode transparency. The same process handles robot movement in both modes, and uses the same movement commands in both simulation and hardware modes. The user can program agents using the most sophisticated movement types available, debug the agents in simulation, and verify the results in hardware without changing anything other than a command line argument. This simplicity reduces the burden on the programmer to handle different modes in software, and increases the realism of the simulation. One of the strongest points of our system is this mode transparency.

For control loops and other time sensitive loops, it is important to separate those loops from other less-time sensitive loops. In the beginning the control function (requiring 30 fps (frames per second)) for each robot was processed in that robot's AI loop, which also performed functions such as path-planning, communications and other less time sensitive functions (10-15 fps). The AI loop could not perform at a high enough rate to satisfy the control laws, and thus the hardware performance of the robots was severely degraded. Isolating the loops allows them to run at their own rate without being slowed by other less time-critical functions.

When possible, we discovered that direct communication between modules is best. This can be complicated with multiple robots, but in our test-bed MCF made it fairly simple. The Referee necessarily monitors inter-agent communication, but all other communication takes place directly between modules and agents. This actually reduces the complexity of communications by allowing for more a logical design and more reusable code. In the early version of the test-bed the Referee had to handle many messages it did not use directly, such as move messages, vision, etc. This increased the possibility of errors during transmission, and made finding and fixing problems more difficult by increasing the number of places in the code a message was looked at. The final system's modules handle only messages they use directly, making for more logical design. Errors can only occur at either end of a point-to-point transmission, making debugging simpler. The flag-client communication object plays an important role in our test-bed by handling these communication details. Removing the burden of communication from the programmer reduces the design time of new experiments.

5.1.1 Future Work

Future work on the test-bed is likely to be limited to minor extensions which increase the flexibility and adaptability of the test-bed for future experiments. These could include changing the Referee so that it can handle arbitrary polygonal obstacles, and more accurately depicting robot shapes. Modifying the Referee to allow it to monitor and enforce communication bandwidth limitations for the agent would allow some interesting experiments. Finally, some documentation to help future users understand the code structure would be an important addition as well. Overall, the test-bed is solid, stable, and fast, allowing it to serve as a useful tool well into the future.

5.2 Safe-Path

We have detailed the important aspects of a good graph generation algorithm: that it takes into account the type and shape of obstacles, scales well both with map size, and the number of obstacles, has moderate memory usage, and has a useful data structure. We have shown two common graph generation algorithms, a Voronoi algorithm and a grid array, and shown why neither alternative suffices in a dense urban environment such as the one modeled here in the MAGICC lab. Finally, we have presented a unique graph generation algorithm, the safe-path algorithm which fills all the requirements of a good graph algorithm as noted above, and shown its effectiveness in actual hardware tests.

5.2.1 Future Work

Future efforts will focus on reducing generation overhead and eliminating the requirement of *a priori* knowledge of the map. We hope to modify the algorithm to allow individual obstacles to be added to the graph as they are found by the robots as they explore their environment. Since most of the generation overhead is due to large number of obstacles that must be checked in different functions, the most likely way of reducing the overhead will involve reducing the number of checks through some sort of intelligent checking mechanism. Such a mechanism would reduce the number of expensive function calls, perhaps by rejecting obstacles based on relative locations.

Eliminating the need for *a priori* knowledge of the map will require removing path segments that cross the new obstacle, and then integrating the new obstacle into the existing map. A simple way to do this would be to regenerate the graph after adding every new obstacle, but as the number of obstacles rises, the cost in computation time would become prohibitive using this method. An ideal method would break the computation time into n smaller chunks, where n is the number of obstacles, and where each chunk is $1/n$ of the generation time of the entire graph. This would allow us to generate the graph as time permits, rather than all at once.

For very large maps this would be a great advantage because the initial generation time can be quite large. Since adding new obstacles will only affect a small part of the graph, breaking the graph into smaller more computationally efficient subsections would further reduce the computation required without affecting the quality of the graph. As the robots explore the map, they would add found obstacles to the graph one at a time, regenerating only a small part of the graph. Initial exploration would require means of navigation other than the path-planner, but once explored, areas could be safely navigated. Such a method would have particular application in military or search-and-rescue type applications in unknown settings, especially in conjunction with a UAV providing visual map updates.

5.3 Experiments

This thesis discussed three studies done using the MAGICC test-bed. Each study utilized the basic infrastructure of the test-bed in different ways. My own study used the test-bed in the 'classic' capture-the-flag mode it was designed for. I used all the capture-flag-rules as originally specified to design an autonomous team of robots. Each agent had its own AI code, running in a separate process. Jeff Anderson's human factors study utilized the same basic structure, while utilizing a set of game rules designed specifically for his work. Finally, Josh Johansen's work utilized a centralized structure with no distributed processing at all. The variety, both in intent and design, of these three studies, demonstrates the flexibility offered by the test-bed infrastructure. While based around a specific game, the test-bed does not force the user into following the rules, instead allowing researchers to create their own structure for their experiments. This flexibility will allow the test-bed a longer, more useful life as a research tool.

These studies had the additional effect of stress-testing the infrastructure, both during and after work was finished. My autonomous team could run for several hours without crashing, freezing or otherwise demonstrating errors in the infrastructure. We achieved this level of stability through hours of testing, debugging, and re-testing. The test-bed is fast, stable, and robust; harmful messages get discarded by the infrastructure before they can cause errors. The entire infrastructure can run on a single Pentium 4 2.4 GHz machine, with two teams of three robots, a UAV, and a basestation without significant slowdown. Finally, since all three studies used the safe-path graph generator and path planner, they served to test its utility too. After generating the graph, the path-planner consistently allows the user to plan paths without any noticeable response delay. In my basestation's case, the user can repeatedly create a new path using the mouse and the agent will plan the path and return it to the basestation as fast as the user can click with the mouse. The planner is fast enough that an agent can plan a new path at every time frame without getting bogged down. In several hours of using it, I have not experienced a crash, unexplainable path, or other error. While anecdotal, this is evidence of its stability and usefulness.

5.4 Conclusion

In this thesis I have presented the design evolution of a flexible test-bed for multi-agent studies. The final test-bed provides a stable, robust platform suitable for a variety of studies, as demonstrated by the differences in the three studies already performed using it. In addition I have presented a unique algorithm for the generation of a graph for path-planning in congested urban environments. Hours of play-testing have shown the speed and reliability of the safe-path algorithm, and it has become the de facto path-planning method used in the test-bed. These tools should provide future students a platform from which they can more easily design and run experiments, reducing the programming overhead and design time of future work, and thereby increasing the efficiency and scholarly output of the MAGICC lab.

Bibliography

[1] W. Ren, R. W. Beard, and J. W. Curtis, "Satisficing control for multi-agent formation maneuvers," in *IEEE Conference on Decision and Control*, (Las Vegas NV), pp. 2433–2438, 2002.

[2] BYU Multi-Agent Intelligent Coordinated Control Labaratory. `http://www.ee.byu.edu/robotics/`.

[3] J. Kelsey, "The magicc mobile robot toolbox (MMRT) : a simulink-based control and coordination toolbox for multiple robotic agents," Master's thesis, Brigham Young University, 2001.

[4] R. D'Andrea and R. M. Murray, "The RoboFlag competition," in *Proceedings of the American Control Conference*, June 2003.

[5] J. A. Adams and A. T. Hayes, "The RoboFlag SURF competion: Results, analysis, and future work," in *Proceedings of the American Control Conference*, June 2003.

[6] M. Blake, R. Falke, and J. Archibald, "MCF, a flexible, robust multi-agent communication framework." Submitted to ICRA, 2002, 2002.

[7] M. Blake, G. Sorensen, J. Archibald, and R. Beard, "Human assisted capture-the-flag in an urban environment," in *Proceedings of the International Conference of Robotics and Automation*, 2004.

[8] J.-C. Latombe, *Robot Motion Planning.* International Series in Engineering and Computer Science; Robotics: Vision, Manipulation and Sensors, Boston, MA, U.S.A.: Kluwer Academic Publishers, 1991. 651 pages.

[9] B. Carter, M. Good, M. Dorohoff, J. Lew, R. L. W. II, and P. Gallina, "Mechanical design and modeling of an omni-directional RoboCup player," in *Proceedings RoboCup 2001 International Symposium*, (Seattle, WA), August 2001.

[10] F. Gentili and F. Martinelli, "Robot group formations: a dynamic programming approach for a shortest path computation," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (San Francisco), pp. 3152–3157, April 2000.

[11] S. J. Russell and P. Norvig, *Artificial Intelligence A Modern Approach.* Prentice Hall, 1995.

[12] S. J. Fortune, "A sweepline algorithm for Voronoi diagrams," 1987. `http://netlib.bell-labs.com/cm/cs/who/sjf/index.html`.

[13] T. W. McLain, "Strategies for the coordinated control of rendezvous of multiple unmanned air vehicles," technical report, Air Force Research Laboratory, Air Vehicles Directorate, September 1999.

[14] P. Chandler, S. Rasumussen, and M. Pachter, "UAV cooperative path planning," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, (Denver, CO), August 2000. AIAA Paper No. AIAA-2000-4370.

[15] H. Choset and Joel, "Sensor-based exploration: The hierarchical generalized Voronoi graph," *The International Journal of Robotic Research*, vol. 19, pp. 96–125, February 2000.

[16] J. O'Rourke, *Computational Geometry In C.* Cambridge University Press, second ed., 1994, 1998.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* MIT Press, 1990.

[18] T. Balch and R. C. Arkin, "Behavior-based formation control for multi-robot teams," in *IEEE Transactions on Robotics and Automation*, 1999.