6-25-2010

A framework for the development of virtual robotic games

Paul Ng

Paul Ng
Candidate

Electrical Engineering
Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dr. Rafael Fierro, Chairperson

Dr. Thomas Caudell

Dr. Pradeep Sen

A Framework for the Development of Virtual Robotic Games

by

Paul Ng

B.S., Electrical Engineering, University of New Mexico, 2007

THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science
Electrical Engineering

The University of New Mexico

Albuquerque, New Mexico

May 2010

Dedication

To my parents, Tang-Tat and Percillia Ng, my brothers, Peter and John, and finally my love, Cassie Ward.

Acknowledgments

I would like to acknowledge my advisor, Dr. Rafael Fierro, who supported me in my journey of accomplishing this thesis. He has always been positive and his spirit has kept me focused and strong willed during the making of this book. I would like to thank him for giving me the opportunity to work with him.

I would also like to thank Dr. Thomas Caudell and Dr. Pradeep Sen, who served on my thesis committee. Their support was invaluable and I hope my research can further their research in collaboration.

I would like to thank all of the MARHES team members. The MARHES team has always supported me, helped me in providing insights, and our fellowship has brought productive cooperation. I would like to specifically thank Damjan Miklic from the University of Zagreb in Croatia, for giving me the initial foundation to begin this work.

Finally, I would like to thank and dedicate this book to my parents, Tang-Tat and Percillia Ng, who taught me from the beginning to enjoy life and work diligently on your works and dreams. I also would like to thank and dedicate this book to my fiancée, Cassie Ward, who also fully supported me and is the most relieved to see this finished.

Paul Ng
Pŋ

A Framework for the Development of Virtual Robotic Games

by

Paul Ng

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science
Electrical Engineering

The University of New Mexico

Albuquerque, New Mexico

May 2010

A Framework for the Development of Virtual Robotic Games

by

Paul Ng

B.S., Electrical Engineering, University of New Mexico, 2007

M.S., Electrical Engineering, University of New Mexico, 2010

Abstract

Multiple robotic experiments are inherently high in operating costs. As the number of agents increase, resources in time, cost, and physical space becomes a limiting factor in experimentation. This thesis presents a framework that addresses these problems by virtual reality. The computer gaming industry has made unparalleled advances in computer graphics and dynamic and kinematic modeling. This progress provides a realism that is utilized in virtual robot experiments with close comparability of results with its real-world counterpart. Developing the Virtual Robotic Games framework with the integration of a video game engine produces a testbed for multi-agent experiments while reducing the cost of real-world experimentation.

The Virtual Robotic Games framework is a platform for the development and experimentation of Robotic Games. Robotic Games refers to a variety of robotic scenarios where autonomous or remotely controlled robots are governed accordingly by the Robotic Game's rules and regulations. These games are a testbed to study robotic algorithms that

are not solutions to only specific scenarios but are applicable to robotic problems of several research topics through a role-based architecture.

The framework distributes Robotic Game algorithms into three leading roles: Game Coordinator, Robot Device, and User Interface. The Game Coordinator is specific to the rules and regulations that must be kept during gameplay. The Robot Device is the player's control algorithms and strategy implementation. The User Interface maintains the human machine interaction, focusing on providing optimal interfaces to ensure a human player can receive the desired results while providing intuitive and simple inputs. This architecture combined with a virtual robotic testbed defines a framework that is rich, extensible, and inexpensive to develop robot algorithms that can subsequently be incorporated into real-world robots.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

In the current world, robotic research is an increasingly expansive field of science and technology. Robots are becoming increasingly capable of diverse and complex tasks. These tasks allow robots to have a spectrum of roles. On one end of the spectrum, manufacturing arms, which provide a large degree of freedom and accuracy to precisely execute path-following and the other, service robots, which interact with humans and cooperatively communicates and maintain relationships with the client. These complex roles are increasingly difficult to study, test, and develop. A solution to reduce the time and cost to prototype current day robots is the use of a virtual simulator [1].

In multiple robot situations, simulators are quickly replacing the cumbersome task of physical experiments. Multiple robot experiments are time consuming and are sometimes expensive in operation. Simulators have evolved to be realistic, rich environments capable of providing accurate simulations that correspond to their real-world counterparts [2]. This capability combined with the flexibility to configure the experiment has made simulators a more feasible option for prototyping. As virtual reality improves, simulators are an

advantageous tool for studying robotic behavior.

Virtual reality is a new field of research in robotics. Physics engines are becoming more efficient and realistic. Modern computers are more advanced and capable of running larger computations faster to model more accurate dynamics and the interactions between rigid bodies. These more efficient simulators provide a reliable testbed that is dependable for prototyping and allows robotic code to be fully immersed in an environment without the expense of setting up physical experiments.

The emerging tasks being developed for robots require high-level algorithms. Scenarios and game schemes are ways to interpret these tasks into a role-based, rule-driven algorithm. Through different scenarios, a cognitive model can be built to develop situational robotic code. Robotic Games provide scenarios with a few objectives to focus the goal and purpose for the robots playing. The Virtual Robotic Games framework will provide a high performance simulator for prototyping and validating the robotic codes used in Robotic Games.

1.2 Thesis Outline

In Chapter 2, the background of the work is discussed. Software framework is defined and broken down to its fundamental purpose. Robotics and several control laws will be explored. The term Robotic Game is defined and the requirements to build a description of a certain game scheme is laid out. The importance of simulators are discussed and its benefits and caveats are weighed. Previous work for these topics are compiled and integrated into the methodology used to build the framework.

In Chapter 3, the framework is discussed. The framework consist of three main modules. These three modules are labeled the Game Coordinator module, the Robot Device module, and the User Interface module. In this chapter, the Game Coordinator's role is

defined and several of its functions are discussed. The Robot Device's role is defined along with robot algorithms and several fundamental middleware processes, which are exemplified. The User Interface's role is defined and its implementation is discussed. This framework can comprehensively construct any robotic game with these main modules.

In Chapter 4, several case studies are shown to demonstrate the framework's capabilities. The first case study is the pilot program, which demonstrates the use of the game manager and the robot device interface and the user interface, which relays robot information through a graphical display. The second case study is a pursuit evasion game, which demonstrates the use of nonlinear control and target-tracking. The third case study is a sharks and minnows game, which involves role playing and game strategizing. The last case study is the sharks and minnows game to demonstrate a comprehensive game containing pursuit-evasion, role playing, and game strategizing.

This thesis concludes in Chapter 5. A discussion is made to address how to develop a game that is appropriate for certain age/skill groups. The framework's performance and limitations will be discussed and future work will be suggested to expand the framework.

Chapter 2

Background

The work presented is done in support of a goal to provide a virtual reality simulator. The field of robotic virtualization is quickly emerging. Though there are many virtual simulators for robotic prototyping, such as RobotiCad [3] and Webots [4], none are focused on the development of Robotic Games. The framework described in this thesis is built to provide an environment flexible to develop robotic games. The basic blocks are already prebuilt in objects and ready to test game schemes. Section 2.1 describes what a framework is and its major traits and purpose. Section 2.2 continues with the definition of robot pose and the kinematics of skid-steered robots. Section 2.3 describes robotic games and what makes them interesting to study. In Section 2.5, the benefits of using a simulator are discussed and the importance of verification. Finally in Section 2.6 previous work in robotic games are discussed.

2.1 Framework

Since the introduction to object-based programming, frameworks have been designed to provide a system for reusing objects on a large scale. It collaborates the increasing number

of objects, managing and cataloging them into classes. Each class describes the different groups of objects. A framework simplifies and creates a consistent procedure to code for complex programming.

There is not an explicit definition for the term software framework, however, every solid framework follows the same purpose [5]. They are used to code for complex technologies. The complexity is broken down into different objects. An object is code that represents an entity. This entity could be concrete or abstract but it's role is defined by the class. The object can be opened and closed at any time by the framework. Objects with similar tasks are grouped into classes.

A class is a set of objects, where all objects have similar traits. These traits are congruent to the behavior of the class type. The class type defines objects to a specific role model and each specific role are equally reusable for a large range of framework contexts.

Several common features that define a framework are:

- Architecture
- Methodology
- Object Wrapping

A framework is an architecture. Its design specifically defines and manages the collection of discrete objects. Its non-modifyable structure provides a foundation that can be implemented repeatedly. Object extensions are attached to the framework to expand or create a more specific application. Since these extensions must follow the framework, objects have consistent style and creates a standard methodology.

The methodology is somewhat reverse to what normally happens in user applications. Usually the program calls libraries or other objects, but in a framework, the flow of control is governed by the framework. This methodology is called the Hollywood principle in

computer programming based on the saying, "don't call us, we'll call you." This decouples objects and reduces dependancies making it easier to test, debug, and maintain. The framework's methodology defines the way objects communicate between each other and how the framework responds to events and data management.

There are distinct differences between architecture and methodology. Architecture is how objects are defined in the framework and methodology is the definition of how objects interact with one another. Defining both of these features in a framework allows us to enforce strict consistency in object communication and function. These objects can be reliably reused for different applications with the same need.

A framework is inherently a wrapper. Since the methodology is already determined, objects must have wrappers to provide a consistent form of communication. A wrapper repackages the object's discrete functions and provides a common interface that is transparent to the other objects. This interface is defined by the methodology of the framework.

Frameworks are designed to provide an environment that simplifies problems through the management of classes, their purpose, and the way they are designed. Classes reduce repetitive coding and decouples object dependancies. Consistency is guaranteed because the framework predefines the classes involved and their methodology among each other. These features help ensure code is productive, reusable and efficient.

2.2 Robotics

Robotics is the engineering science pertaining to robots, their manipulation, and their automation techniques [6]. Its broad field includes the design and manufacture of robots, such as the development of new robots with new articulation, BigDog [7], Robotic Hands [8] or reducing the manufacturing cost through industry [9]. Robotics also includes the software and algorithms that control its decisions and movements. This is equally as large,

encompassing trajectory generation, human interaction, signal processing, image processing, and robot networks; such as formation control, swarming, cooperative agents, and robotic games.

The robot's mechanics depend on the manipulator used. Most actuators use force to drive motion into the robot. Motors and joints require torque inputs but returns its angle and angular velocity. This requires the use of dynamics to calculate a desired trajectory. Wheels have odometers to calculate displacement. Modern robots are now equipped with laser range finders, sonar, global positioning systems, and camera vision to increase awareness of the robot's environment.

The notation to describe the spatial frame can be found in [10]. An object or robot can be described as a point in space. This position can be written as a tuple where each value is the coordinates to the Cartesian orthogonal frame. In Figure 2.1, there is a fixed reference frame where the origin for that frame is located. The tuple is insufficient to describe the robot completely. The term pose describes not only the position of the robot but also its orientation. In order to describe its orientation, a frame to the robot is attached.

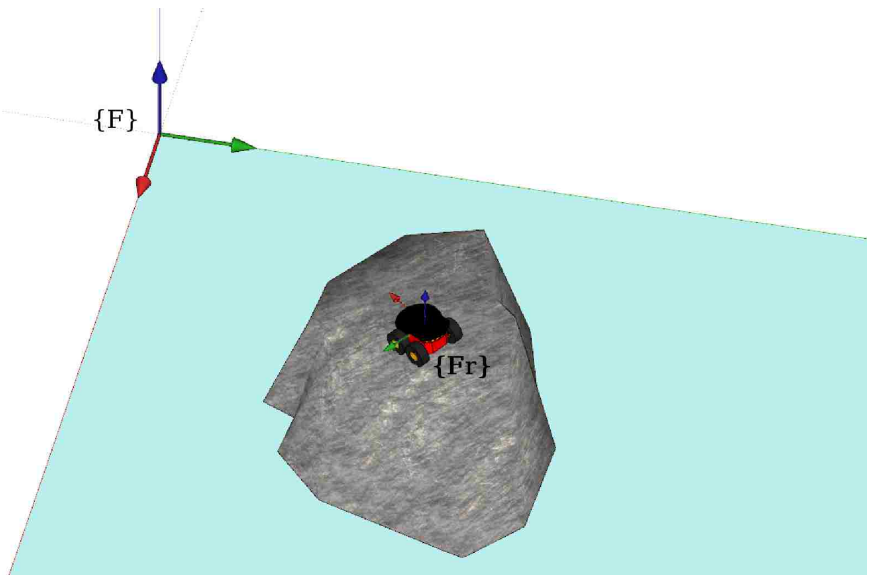

Figure 2.1: The Reference Frame F and Robot's Frame F_r.

The frame is attached such that the $\hat{X}_r$ is pointing forward, the $\hat{Y}_r$ axis is pointing port side, and the $\hat{Z}_r$ axis is pointing up. Since the robot's frame is defined and fixed, the object's orientation can be described by the Z-Y-X Euler angles. In Figure 2.2, the pitch, roll, and yaw are graphically shown. The name Z-Y-X describes the order in which the rotations are done. Starting from the reference frame's orientation: the frame is rotated about the $\hat{Z}_r$ by an angle α, then the frame is rotated about the $\hat{Y}_r$ by an angle β, and finally the frame is rotated about the $\hat{X}_r$ by an angle γ. The final frame represents the orientation of the robot frame. The angles α, β, and γ are equivalent to the robot's yaw, pitch, and roll respectively.

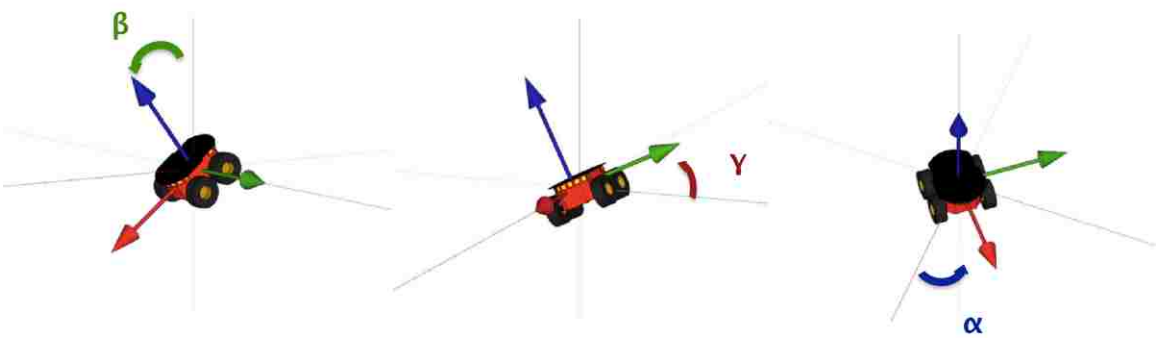

Figure 2.2: Pitch (about the y axis by angle β), Roll (about the x axis by angle γ), Yaw (about the z axis by angle α).

For skid-steered robots, the kinematic model assumes four major assumptions: the mass of the robot is at the geometric center, each side's wheels rotate at the same speed, each wheel has a point contact to the ground, and the robot's four wheels are always on the ground. For the motion dynamics, the control scheme is sufficient in two dimensions. In Figure 2.3, the skid-steered robot's model is graphically shown. The position of the robot is defined by the reference frame F, where the $\hat{Z}$ component (elevation) is neglected. The angle θ is the rotation about the $\hat{Z}$ axis and is equivalent to its α component or yaw. The

kinematic model is approximated by the unicycle model [11]:

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{\theta} &= \omega, \end{aligned} \tag{2.1}$$

where the robot's input is its linear and rotational velocities, $u = [v \quad \omega]^T$. The distance d is an arbitrary point off the robot's center of mass to avoid singularities.

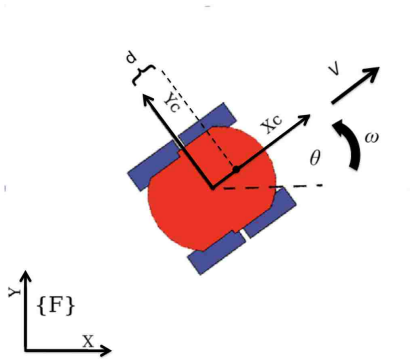

Figure 2.3: Plot of a skid-steered robot.

The robot frame is used to localize the dynamics controller to each robot and the point d is used for the following outputs:

$$\begin{aligned} y_1 &= x_c + d\cos\theta, \\ y_2 &= y_c + d\sin\theta, \end{aligned} \tag{2.2}$$

where $d \neq 0$.

Taking the time derivative of Equation(2.3) computes the Matrix $\mathbf{T}(\theta)$ (Equation (2.3)).

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -d\sin(\theta) \\ \sin(\theta) & d\cos(\theta) \end{bmatrix} \begin{bmatrix} \dot{x}_c \\ \dot{y}_c \end{bmatrix} = \mathbf{T}(\theta) \begin{bmatrix} \dot{x}_c \\ \dot{y}_c \end{bmatrix}, \tag{2.3}$$

Therefore the Matrix $\mathbf{T}(\theta)$ is invertible to provide the robot's inputs as:

$$\begin{bmatrix} \dot{x}_c \\ \dot{y}_c \end{bmatrix} = \mathbf{T}(\theta) \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \tag{2.4}$$

which can be applied to Equation (2.2) in the form of

$$\begin{aligned} \dot{y}_1 &= u_1, \\ \dot{y}_2 &= u_2, \\ \dot{\theta} &= \frac{u_2\cos\theta - u_1\sin\theta}{b}. \end{aligned} \tag{2.5}$$

A simple linear controller can be derived (Equation (2.6)).

$$\begin{aligned} u_1 &= \dot{x}_g + k_1(x_g - y_1), \\ u_2 &= \dot{y}_g + k_2(y_g - y_2), \end{aligned} \tag{2.6}$$

where $k_1 > 0, k_2 > 0$ guarantees convergence to zero of the cartesian tracking error. This trajectory tracking is based on output error and is an input/output feedback linearization algorithm [12], depicted in Figure 2.4, where G^{-1} is the transform from the input u to the robot's motor inputs: v_c and ω_c.

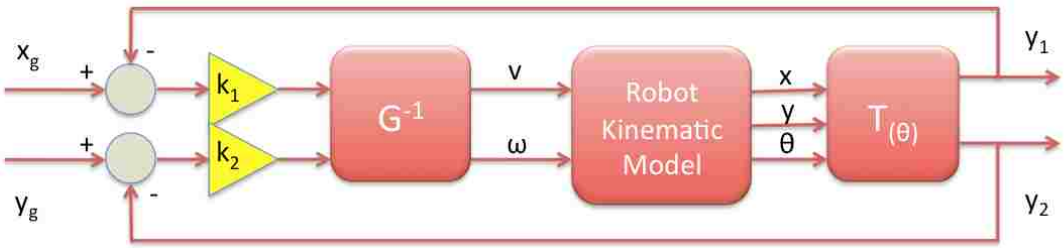

Figure 2.4: Feedback Linearization.

2.3 Robotic Games

A robotic game is an environment where the robot's decisions and movements are governed by a predefined set of rules. Any robot who is governed by these rules is called a *player*. The rules include a scoring scheme and the goal of the *player* is to get the best score. Usually there is an opposing team and strategies are required to keep the advantage. There is also a constraint in time so promptness must be considered in the strategy. The game can be autonomously controlled if two things are addressed.

A robotic game can be described in two main parts:

1. Rules
2. Strategies

The rules are upheld through many ways but are most commonly managed by a game *referee*. A *referee* oversees the gameplay and maintains fair playing conditions. It holds the official game clock and tracks penalty times. The score is kept by the referee and score penalties are appended when necessary. It calls players if misconduct has been detected, however, some penalties may be overlooked if it disrupts the flow of the game and the penalty was not significant.

The referee does not impose the rules onto the players rather watches the players during the game. The referee's calls are not disputed by the players but could be disputed by the team's captain. A third character titled as the official would give the final call to veto the referee. The game's rules are required to keep the game interesting, fair, and well balanced, providing an environment for the derivation of strategies.

A strategy is an option a player may take to achieve a certain goal. In a cooperative game, all the players decide and may perform multiple strategies to optimize their score in the game. In a non-cooperative game, opposing teams have opposing goals and may exhibit Game Theory optimization. In Game Theory, two kinds of strategies exists: mixed strategies and pure strategies. Mixed strategies are strategies that can be randomly picked during a certain situation in the game. Pure strategies are causal, requiring a certain move from the opposing team. These strategies lead into a branch of Game Theory called Combinatorial Game Theory.

Robotic games are environments where robots are players with a set of rules. These rules are predefined and a certain goal is required from the players. A referee oversees the game, ensuring fair gameplay and is also the timekeeper and scorekeeper. An autonomous robotic game requires the strategies of the players it controls. The player's strategies can be mixed or pure strategies. Forms of difficulties may be a set limit to the number of robots on the field or a set time limit. Robotic games provide the challenges and excitement for all ages to encourage interests in science, engineering, and robotics.

2.4 Finite State Machines

Finite state machines are models of robot behavior in a finite set of states and a finite set of transitions between each state. This model allows study in hybrid and switched systems, where the control scheme has both logic-based and continuous dynamic systems [13]. Finite state machines follow the same rule set as robotic games and are natural in modeling the dynamics of a Robotic Game.

The remainder of the thesis will follow the CHARON notation [14]. The CHARON specification defines application structure, architecture, and behavior. CHARON state diagrams differ from a flowchart in that flowcharts describe the transition from one state to another while CHARON state diagrams describe the states, Figure 2.5. CHARON requires explicit trigger events while flowcharts may not have any events. This is important as the CHARON state diagram will require the work in this thesis to rigorously determine all possible events.

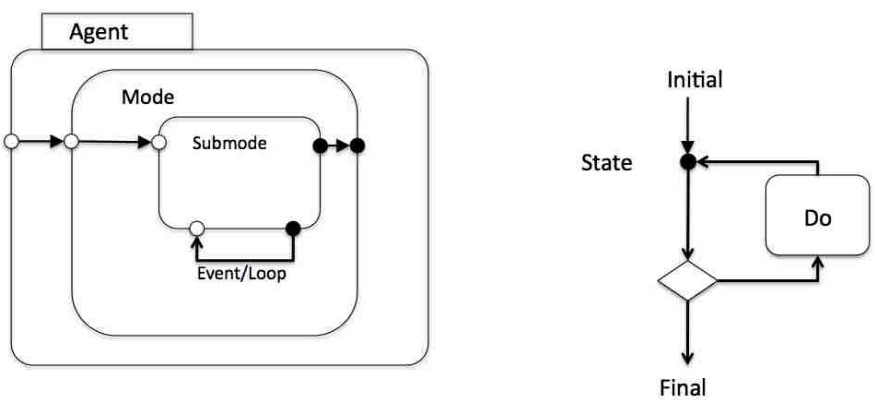

Figure 2.5: A comparison between CHARON state diagram (left) and a flowchart (right).

The CHARON state diagram is similar to a Harel statechart [15] with a specific notation. CHARON is a description of the architectural and behavioral hierarchy of the

software. From Figure 2.5, the communication and shared variables are inputs and outputs to the rounded rectangle labeled Agent. Agents can represent a robot, object or a referee. An experiment or game can have any number of Agents. An Agent consists of several embedded *Modes*. *Modes* represent a certain strategy or complex operation. *Modes* are comprised of a single or multiple set of behaviors each referred to as a *submode*. *Submodes* contain the dynamic control scheme.

Inputs between hierarchical layers are denoted by *ports*. A hollow circle represents an input either from another block or a higher level block. Outputs are represented by a solid circle which may recursively return as an input such as the *Mode* block or passed as an output of a higher level block such as the *submode* block. These ports are governed by a transition and a trigger event. The event is denoted by the text near the transition arrow. The arrow labeled as Event when triggered runs the Loop code.

The CHARON specification is an accurate representation of hybrid systems based on behavioral and architectural design. CHARON conforms to the nature of multiple robot experiments and provides a notation that allows an accurate graphical description of robot behavior based on conditional transitions. CHARON has been applied to multiple robot coordination in mapping, localization, target tracking, and formation control [16].

2.5 Simulator

The use of simulators to develop prototype robotic code is becoming paramount. As robotic hardware can be large in capital cost, the use of simulators can drastically reduce the cost and time to develop prototype code. Providing the requirements necessary for a simulator to have accurate results. Simulators can also be used for educational purposes since they are more portable and more forgiving than physical experiments.

The authors in [17] describe a number of requirements for simulator validation. A

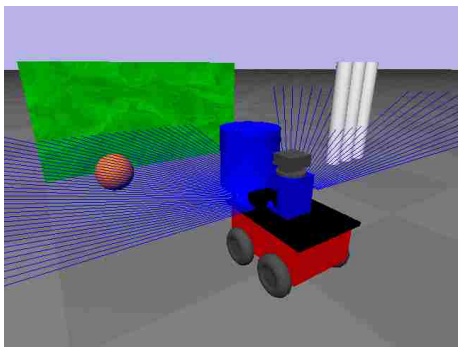

Figure 2.6: Gazebo: Player/Stage's 3D simulator with dynamics.

simulator's physics engine is the heart of the simulator. The robot's movements are determined through the simulator's rigid body dynamics. Physical dimensions and dynamics of the environment, robots, and their interactions must be modeled to the highest accuracy to reduce simulation errors. Though the simulator's spatial computations are preeminent, the second feature that is required is their visual realism.

Figure 2.7: Cyberbotics Webots: commercial 3D robot prototyping and simulator.

The visualization of the environment and the containing robots can be displayed in a two dimensional graph. Laser finders, sonar, and odometry sensors are planar and can be accurately modeled as long as the environment is also in the same two dimensions. The implementation of three-dimensional visualization increases the capability for cameras and image processing. Three dimensions also allows for experimentation of different

poses, adding both pitch and roll to the robot.

A simulator must be flexible. Models of different robots, actuators, and sensors supply an arsenal of robots with a vast combination of attachments. Models of new robots and actuators can be added to continually expand the simulator's set. The environment can be carefully modeled to have a direct replica from reality to virtual reality. The robot controller code can be designed to provide code transferable for both the virtual robot and the physical equivalent robot. The reduction of time to port the code will increase the productivity of the simulator.

Figure 2.8: Pioneer 3-DX.

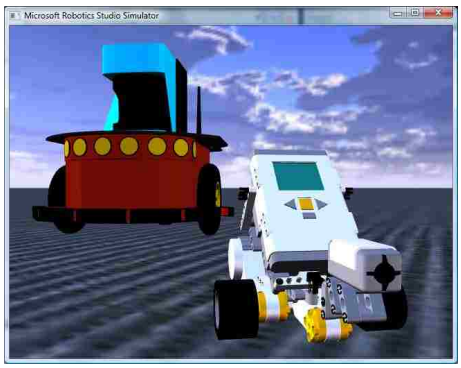

Figure 2.9: Microsoft Robotic Studio: robot models and simulator.

The simulator must be efficient. The simulator must run as close as possible to full frame rate as lag in simulation leads to many different errors. Its physics engine renders collisions at full frame rate and a large number of collision boundaries will render jitters. The dynamics controller for the simulated robots requires instantaneous sensor readings to calculate its input u and delayed values generate an overshoot to the desired capture point. Simulations of sensors must be instantaneous to simulate the behavior of the physical sensors' values.

Simulators are becoming popular due to their cost, comprehensiveness, and accuracy. However, without the highest of accuracy in modeling, the simulator over-simplifies the

Figure 2.10: UNM's Centennial Engineering Center.

Figure 2.11: USARSim: accurate elaborate environments.

physics of the models and provides an ideal solution to a non-ideal world. As computers advance, simulations are becoming increasingly capable and this problem is quickly diminishing. The use of simulators goes hand-in-hand with robotic games due to their video game like environment. Its portability and large forgiveness for errors makes it optimal for prototyping robotic code.

2.6 Related Work

Robotic Games began as early as four decades ago. One competition during this period was the Micromouse [18]. A robot is positioned somewhere in a 10' x 10' field and is required to find the center of the maze. Since then, the competition has advanced to a larger field of 16' x16' square, however, the strategy solution is still considered the same. Other more complex games have been developed to broaden the excitement of Robotic Games [19].

Problem scenarios began to appear including the Lion and Man problem [20], Marco Polo [21], Apollonius Pursuit [22], target-tracking[23], motion planning [24] and the RoboFlag Competition [25]. These scenarios have been involved in robotic games and

have encouraged research to develop solutions for robots. The Lion and Man problem is a pursuit-evasion scenario where a robot may be in the pursuit or evasion from a person or robot. The pursuer must catch the evader and the evader must keep its distance. Marco Polo is a follower-leader scenario where a patrolling robot may need to observe an intruder. The authors in [26] studied several problems regarding guaranteed strategies for patrol agents. These scenarios evolved to incorporate more complex situations and soon robotic competitions of grander scale began to appear.

The RoboFlag competition is a complete robotic game. Based on "Capture the Flag," it consist of two teams, each required to take the opponent's flag and returning it to home base while protecting their own flag. These encompasses multiple problem scenarios to add complexity and required solutions that were more unpredictable and similar to real-world situations. One of the largest current competitions is the RoboCup [27]. RoboCup is an annual competition where autonomous robots compete in a game of soccer. The competition is broken down to different classes sorting robots by their relative size, capability, and form of mobility. The dynamics of soccer is generally researchable due to soccer's popularity and enthusiasts. Strategies used by soccer players can be extended to robot players. The authors in [28] demonstrate a RoboCup strategy and have validated it through match data. RoboCup player algorithms have also improved in the areas of ball passing [29] and other algorithms for specialized platforms such as Segways [30].

These topics of research have also encouraged research in simulations [31]. Gazebo[32], an open-source robotic simulator backend for Player/Stage[33], capitalizes on the open dynamic engine (ODE) for computing dynamics. Though the open-source is appealing, Gazebo runs on older libraries, which limits its capabilities in multiple robot simulation. The commercial Webots [4], is a high-quality simulator focused on accurate modeling of actual popular robots. It is a fast, easy to use simulator but is not free or modifiable. USARSim is similar in scope to Gazebo, Webots and Microsoft Robotics Studio. USARSim's largest feature is its use of a video game engine [34]. It uses the Unreal game engine to

produce accurate dynamic models and support large worlds.

USARSim has been used as a testbed for multiple robotic research. This research spans across many topics of robot research[35] including Robotic Games. Several major projects using USARSim are MAST[36], NIST's MOAST[37] and DARPA's SyNAPSE[38]. USARSim additionally participates in several competitions such as RoboCup[39] and IEEE Virtual Manufacturing Automation Competition[40]. Robot prototyping has been done with USARSim for simulations of the AIBO Class players in RoboCup [17]. The CHARON notation has been applied to multi-agent hybrid systems [41] and in multi-agent coordination [16] and provides a specification for building complex agents through a hierarchical behavior composition. These research topics have provided the design characteristics needed to develop the Virtual Robotics Games framework.

Chapter 3

The Framework

3.1 The Virtual Robotic Games Framework

As depicted in Figure 3.1, the framework consists of three modules to comprehensively simulate a game. The Game Coordinator coordinates all game strategies and manages high level decision making. The Robot Device module bridges the high level decisions made by the Game Coordinator to the robot manipulation. The User Interface module provides user inputs from a graphical interface or joystick to the framework. Configurations are discussed demonstrating the flexibility and modularity of the framework's three modules. These same characteristics were also used when implementing the framework into software. This chapter is a description of a developed, high depth virtual simulator that allows simple implementation of a robotic game.

3.1.1 Game Coordinator

The Game Coordinator module is the heart of the framework. It initiates instances for each Robot Device and User Interface. Its role is comprised of virtually setting up the

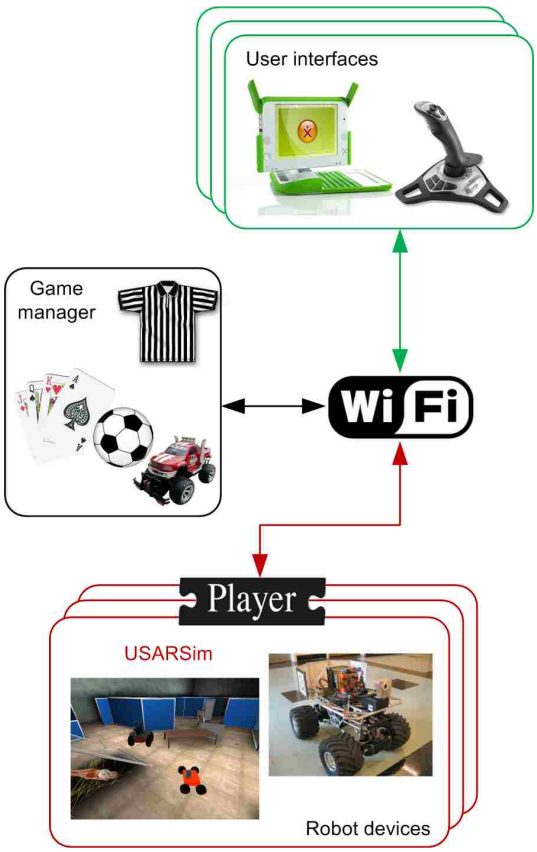

Figure 3.1: The Virtual Robotic Games framework.

game scenario. As defined in Section 2.3, a Robotic Game requires rules and the Game Coordinator maintains these rules. The Game Coordinator does not take control of any of the other modules but rather establishes their relationships and sets up policies for their interaction in the Robotic Game.

The Game Coordinator module has two roles. The first role is to define the game and ensure all players of the game are present. The second role is to determine the game scenario and execute the game scenario functions. The Game Coordinator was given these roles because much like a referee in a sports game, it ensures that both teams meet initial conditions on the field and oversees fair gameplay.

The Game Coordinator initiates automatically after a few user inputs determining the environment and the game to be played. It initiates the players through the Robot Device module and introduces the user to the game through the User Interface module. During the game, it cycles through a set of rules defined by the chosen game and performs its duties when necessary.

During the event of a gameplay change, the Game Coordinator alerts the Robot Device modules and any graphical user displays. A graphical display can be constructed at the appropriate time relaying information regarding the game and its players. This helps debug certain situations effectively and systematically.

3.1.2 Robot Device

The Game Coordinator initiates one Robot Device module per player. Each Robot Device is configured to a certain robot and their task in the game. Its role provides the player's robot automata and a platform abstraction that allows the use of a broad application of robot devices. The Robot Device module is the essence of the robot players in the Robotic Game.

The Robot Device module, at the least, contains an initialization and shutdown processes. These processes are specific to a virtual or physical robot configuration. The initialization process connects to the robot and opens the communication socket to control and receive data. A virtual robot setup also includes placing the player into the game. The shutdown process does the inverse. The shutdown process sends the robot a halt signal, stopping the motors and terminating the sensors before disconnecting and, if applicable, removing the player from the virtual world.

Robotic code is stored in the Robot Device module. The Robot Device module contains a hardware abstraction layer for flexibility and maintains compatibility with other robotic platforms. This layer sends wheel velocities and arm angles while receiving sen-

sor data and hardware status from the robot. High-level algorithms connect to the interface provided by the hardware abstraction layer to implement path planning, high-level computations, and other game strategies.

3.1.3 User Interface

The User Interface module handles human interaction with both the Virtual Game and the Virtual Robotic Games framework itself. The module can be broken into two parts. Much like the Robot Device module, the User Interface module contains a hardware abstraction layer that provides a unified interface for high-level processes. This interface consists of interpretations such as move, strategy calls, and communication. The high-level processes receive these interpretations and apply them to their function. One function might be to control and command a player robot. The user can control the player's movement, tell the other players its status, and suggest a strategy.

The User Interface module returns information to the user through a graphics display. Providing the user's player game data through a display, the user can analyze the situation in a live game without decrypting lines of text. Graphics displays are also used in the development of Robotic Games. The status of all the players and the status of the Game Coordinator can be displayed. Omniscience data provides the necessary detail for proper evaluation of a situation and can help explain glitches and loopholes in game development and player strategy.

The Virtual Robotic Games framework requires the use of an User Interface module to provide an interactive, efficient environment for developing Robotic Games. The User Interface module allows users to watch from the control center to ensure robustness in games. Game situations and robot strategies are tested and validated to ensure proper behavior development. A graphic display can also play a role as the head-up display (HUD). An intuitive display can effectively allow the user to realistically feel like he is

operating the robot player and become engrossed in the virtual robotic game.

3.2 Software Implementation

The framework uses Mathworks MATLAB with the combination of USARSim to bring an extensible high-level algorithmic computing environment with realistic three-dimensional physical simulations. The integration of these two products provides an interactive cross-platform application effective for the development of Robotic Games.

3.2.1 Mathworks MATLAB

MATLAB is a numerical computing environment that is one of the de facto among industry and academic worlds. The MATLAB programming language is a powerful language rich with extensible toolboxes, object-oriented programming, and graphical user-interface development (Figure 3.2). It is available for Microsoft Windows, Apple Mac OS X, and Linux platforms and only very slight alterations are needed to port code between operating systems.

MATLAB accommodates Model-Based Design and simulation in an environment called Simulink. Simulink is a graphical control block diagram simulator. Block models are placed in the simulation and signal parameters are linked between the blocks. Simulink is a graphical interactive simulator that provides a flow chart environment fully integrated with MATLAB's processes, however, Simulink is not currently implemented in the framework.

The MATLAB programming language allows programmers to program in a high-level language. This language looks much like C++ however is functioned to be a mathematical scripting language. MATLAB stores scripts as m-files and m-files may contain a function,

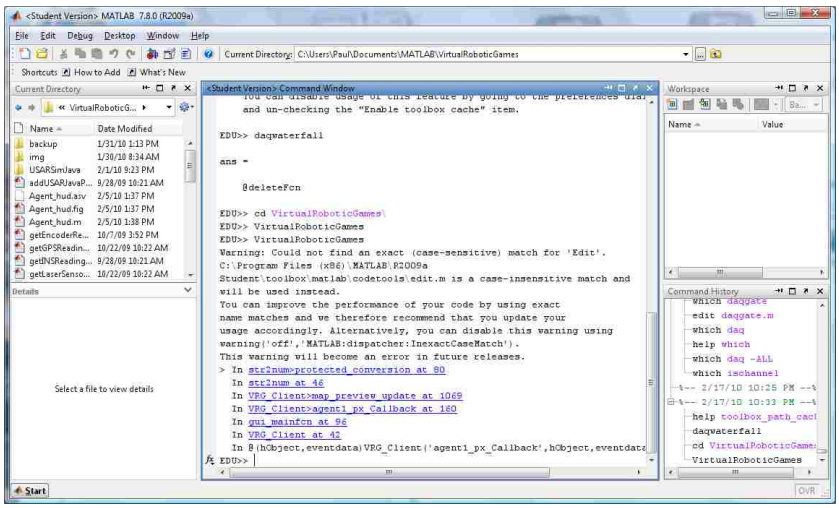

Figure 3.2: The MATLAB Environment in Windows Vista x64.

class definition for object-oriented programming, or graphic display. The framework uses all three types of m-files. All algorithms in the framework are programmed as functions in m-files. Functions consist of input and output arguments and arguments in MATLAB are pass by value.

Due to the control flow of the Game Coordinator module, conditional functions are heavily used to proctor the game scenario. Game rules must be explicitly described in conditional expressions. Also information of the game must be provided and this is determined through conditional expressions to narrow down the game status. A generic Game Coordinator algorithm is provided in Algorithm 3.1.

In MATLAB version 7.6, object-oriented programming is introduced. This is extremely resourceful for developing code that could have multiple instances such as the Robot Device module in the framework. This allows programmers to build a class that could represent certain virtual entities such as the role of each player. Objects improve data management, software complexity, and code reusability.

Algorithm 3.1 Game Coordinator Algorithm

```
Perform initial robot placement
while Game_Status = True do
    Send Game_State ← Initialize to Player Objects
    while Initial gameplay conditions are not present do
        Update Game Object's Positions
    end while
    Send Game_State ← Play_Round to Player Objects
    while Game_State = Play_Round do
        if Foul conditions are present then
            Game_State ← Foul
            Execute Foul Algorithm and Adjust Score
            while Game_State = Foul do
                Send Player Objects to game resumable state
                if Gameplay conditions are present then
                    Game_State ← Play_Round
                end if
            end while
        end if
        if Scoring conditions are present then
            Adjust Score
        end if
        if Time or Gameplay conditions suggest Round completion then
            Game_State ← Round_Over
        end if
    end while
    if Time or Gameplay conditions suggest Game Over then
        Game_Status ← False
    end if
end while
```

MATLAB's terminology for object class definitions are described as below:

Classdef	Code description of the object class.
Properties	Variables that are common to each object instance.
Methods	Every object instance's member functions.
Attributes	Values that defined properties and methods.

The different roles in a Robotic Game follows the concept of object-oriented programming very effectively. The initialization code required to attach a player to the game is constructed in the object's constructor and similarly, when the player is to be removed from the game, its termination code is placed in the de-constructor. This ensures good housekeeping and helps build a tight program.

MATLAB defines an object's class with a class definition file similar to Algorithm 3.2. The class properties hold data items regarding a certain class instance and the class methods are the object's specific member functions. Both class properties and class members are only used within the specific object. None of the properties or the methods contain any information that is required by the Game Coordinator or by another object. The object serves its purpose as a stand-alone entity and is removed from any dependencies.

The class properties contain their object's details such as its position, sensor readings, and game status. From a simple goal seek to strategic computations, the functions used in each player can only use their own information from their sensors and the information provided by the Game Coordinator. The Game Coordinator then becomes the communication link and allows programmers to work in a decentralized manner.

MATLAB is an interactive environment where programming, computation, and data analysis combine together to create a complete development platform perfect for designing new robotic games, robotic algorithms, and validation. MATLAB also includes visualization, which fits perfectly to the description of the User Interface module. MATLAB provides GUIDE as a way to simplify the User Interface development.

Algorithm 3.2 Robot Player Object

```
classdef Agent < handles
    % The object class Agent is an inherit function of handles
    properties
        % Object's data members
        agent_obj           % Object file
        state               % Agent's Status
        laser               % Laser Readings
        sonar               % Sonar Readings
        odom                % Agent's Pose
    end

    methods
        % Object's member functions
        function obj=Agent(name, type, start_pos, start_ori)
            % Constructor
            agent_obj = initializeRobot(name,type,start_pos,start_ori)
        end
        function state = getAgentState(obj)
        function laser = getLaserReadings(obj)
        function sonar = getSonarReadings(obj)
        function odom = getPose(obj)
        function goto(obj, goal_x, goal_y)
        function delete(obj)
            % Deconstructor
            shutdownRobot(obj.agent_obj)
        end
    end
end
```

MATLAB's Graphical User Interfaces Design Environment (GUIDE) is a development tool to create and interactively edit graphical displays. Graphical User Interfaces (GUIs) are saved as a FIG-file and a M-file and serve as the User Interface module in the framework. GUIs simplify tasks and allow users to accomplish the same tasks without fully understanding the commands and syntax to perform those tasks. To the framework, the GUI corresponds to MATLAB functions and provides event triggers or input values. This integrates the User Interface module nicely with the other modules.

GUIDE is an interactive visual GUI developer (Figure 3.3). Built-in components can be dragged and dropped to populate a FIG-file to the programmer's desire. User interactions with each component executes a callback. Programming the appropriate task to each callback allow the GUI to be rich, simple, and intuitive. Several of the important components are push buttons, radio buttons, text boxes, menus, and most importantly axes.

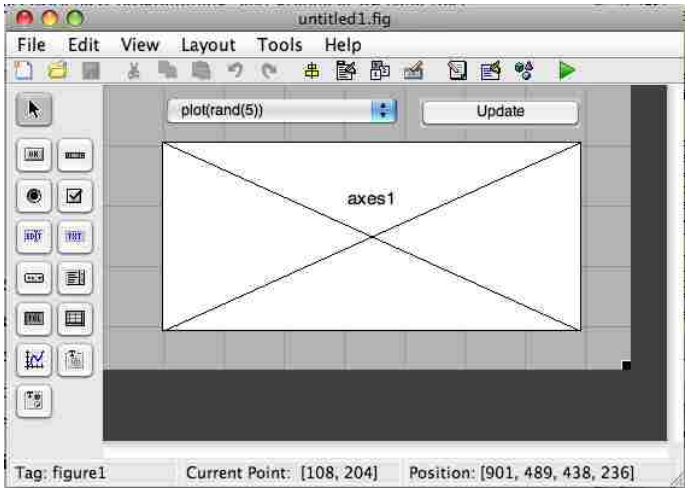

Figure 3.3: The GUIDE environment.

The biggest strength for Matlab is its extensible platform. Combined with a large community, MATLAB has an expansive dynamic library of addons called toolboxes. Toolboxes are packages of m-files created either in-house or by an outside programmer. The framework uses two toolboxes not provided in the student version: Simulink 3D Anima-

tion Toolbox and Drexel University's MATLAB USARSim Toolbox.

The Simulink 3D Animation Toolbox provided by MATLAB is a complete package for developing 3D visualization and building connections to hardware input devices. These devices include sound cards, data acquisition hardware, and most importantly for this framework, usb joysticks and gamepads. In the framework, the toolbox is utilized to provide user input from a gamepad shown in Algorithm 3.3. Gamepads are ergonomic, intuitive, and equipped with a large amount of axes and buttons. The toolbox is fully integrated with MATLAB and contains Simulink blocks to plug data entry directly into Simulink models.

Figure 3.4: The Joystick Input block from MATLAB's Simulink 3D Animation.

Drexel University's MATLAB USARSim Toolbox[42] is the bridge that connects MATLAB to USARSim and Unreal Tournament 2004. Following the gamebot protocol[43], the toolbox provides methods to send drive commands and receive status and sensor readings from the robots in USARSim. The toolbox connects to USARSim as a client and initializes robots through the protocol detailed in Appendix B.

MATLAB combined with the USARSim toolbox is the controller that governs the robots in USARSim. This constructs the majority of the Virtual Robotic Games framework. The Game Coordinator module is built in a conditional function. The User Interface module is provided through a GUI developed in GUIDE and users provide gamepad inputs through the daq toolbox. The Robot Device module is the collaboration between the robot's computational algorithms and its virtual robot entity. USARSim provides the virtual environment and the player's virtual robot bodies.

Algorithm 3.3 Joystick Object

```
classdef Joystick < handles
    % The object class Joystick is an inherit function of handles
    properties
        % Object's data members
        joystick_obj            % Joystick Obj
    end

    methods
        function obj=Joystick(id)
            % Constructor
            joystick_obj = vrjoystick(id)
        end
        function state = getJoystickState(obj)
            [state.axes, state.buttons, state.povs] = read(obj.joystick_obj);
        end
        function delete(obj)
            % Deconstructor
            close(obj.joystick_obj)
        end
    end
end
```

3.2.2 USARSim

The National Institute of Standards' (NIST) Reference Test facility maintains USARSim. The Unified System for Automation and Robot Simulation (USARSim) was designed for

urban search and rescue simulations and have now grown to include many ground vehicles, the DARPA urban challenge, robotic soccer, submarines, unmanned aerial vehicles, and humanoids. The USARSim's mission is to provide a high quality simulations that supports accurate user interface behaviors such as camera video, sensor readings and accurately portray robot automation and behavior.

USARSim's architecture is depicted in Figure 3.5. USARSim consist of a modification to the Unreal Tournament 2004 (UT2004) engine. Robot, sensor, and map models are inputted and compiled into UT2004. The UT2004 networking is proprietary, however, with the University of Southern California's GameBots [43] a protocol allows client controllers access to characters in UT2004 via network sockets. The MATLAB USARSim toolbox connects to UT2004 through this protocol. The Unreal Client is used to generate video from the game. Using the multiview feature, multiple robot cameras can be captured and displayed on the screen. An image server can grab the video from the Unreal Client through a DirectX hook.

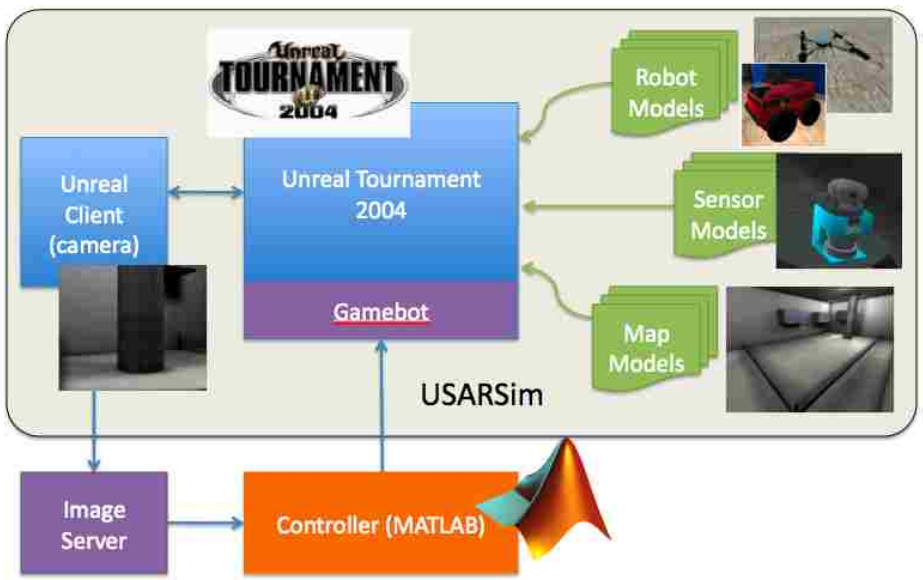

Figure 3.5: The USARSim architecture.

UT2004 is a commercial video game that provides a stable modern physics engine and is open to gamer modifications (game mod). UT2004 is a deathmatch-type first person shooter video game developed by Epic Games. The Unreal series began in 1998 and in 2004, UT2004 became the sequel to Unreal Tournament 2003. UT2004 and its engine, Unreal Engine 2.5, is cross-platform, available for Microsoft Windows, Apple Mac OS X, and Linux x86-32 and x86-64 bit versions.

The Unreal Engine 2.5 is a modern physics engine integrated with the Karma physics SDK. This powerful physics engine delivers realistic rendering performance, collision detection, vehicle physics, particle systems, networking, and file system management in one package. The engine is built primarily for graphical realism and smooth gameplay. Epic Games does not release its engine source but has developed an Unreal scripting language, which allows the inclusion of modifications. With this capability, robot researchers can tap into Unreal Engine's power and develop a research tool for simulation and prototyping.

USARSim's Robot models account for the specific robot's configuration. If the robot is a two wheeled differential drive vehicle, the robot's chassis dimensions, trackbase and wheel radius are computed with the individual torques to each wheel to determine the robot's movement. The sensors and effectors are also mounted in a defined position and it outputs specific properties such as maximum range, resolution, and readings. Three dimension models and maps can be built using Epic Games' Unreal Editor 3.

The Unreal Engine uses a Left Handed Coordinate system represented in Figure 3.6. To apply a standard, USARSim uses the SAE J670 Vehicle Coordinate System depicted in Figure 3.7. The Unreal Engine also uses their own unit system called an Unreal Unit (UU). USARSim also standardized that 250 UU is equivalent to 1 meter and 65535 UU is equivalent to 360 degrees. The framework also uses USARSim's standardization in both coordinate system and unit conversion.

USARSim is a flexible UT2004 modification rich with prebuilt robot platforms and

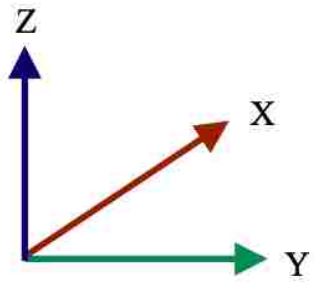

Figure 3.6: The Left Hand Coordinate System.

sensor models. With fully customizable robot configurations and maps, USARSim is extensible to a variety of robotic scenarios. Powered by the Karma physics SDK, the Unreal Engine brings an authentic reality to robot research. USARSim combined with MATLAB as a controller completes the Virtual Robotic Games Framework as a full, realistic, computational environment suited for the development of Virtual Robotic Games.

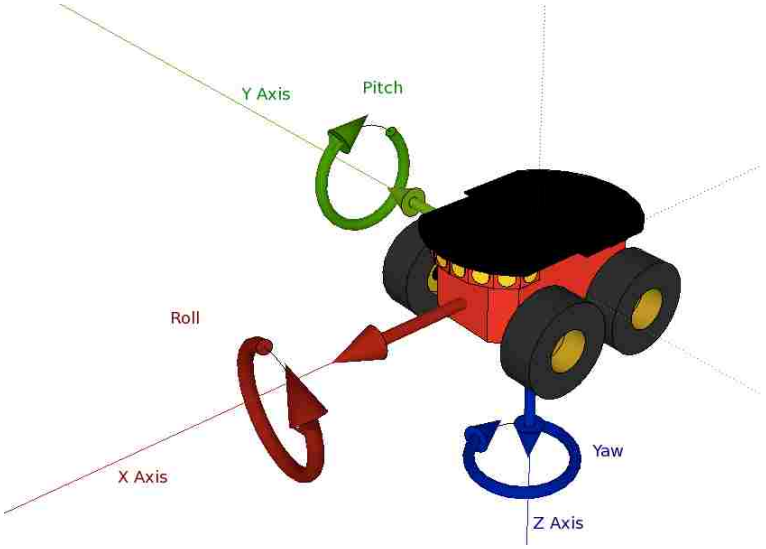

Figure 3.7: The SAE J670 Vehicle Coordinate System.

Chapter 4

Case Studies

The following Robotic Games were developed using the framework discussed in Chapter 3. The Framework was developed on a Windows Vista platform using an Apple Macbook Pro 3.1, however, both MATLAB and USARSim are cross-platform and the framework could be implemented onto Mac OS X and Linux easily. It is suggested to have at least a dual-core system as USARSim's server application tends to devote the cpu to itself.

4.1 Player Head-Up Display (HUD)

The Player Head-Up Display is a pilot experiment game. The goal of this game is to display a robot dashboard providing information regarding its location, sensor readings, and a simple goto function that utilizes the algorithms from Figure 2.4. There are no rules in this game. The Game Coordinator has one task and it is to construct a GUI for every robot. Each GUI will have an open robot button and several axes, each containing information regarding location, laser readings, sonar readings, and a goto panel.

The GUI has been built to help users start USARSim and pick an environment. The

GUI is shown in Figure 4.1. A list of predefined maps are shown and when the user selects a map and clicks the Start button, the GUI initializes the USARSim environment with the chosen map. A separate USARSim spectator window appears on the screen. This window is a spectator camera and can view the game from an outside point of view. The user must switch windows back to the Virtual Robotic Games framework to select the players.

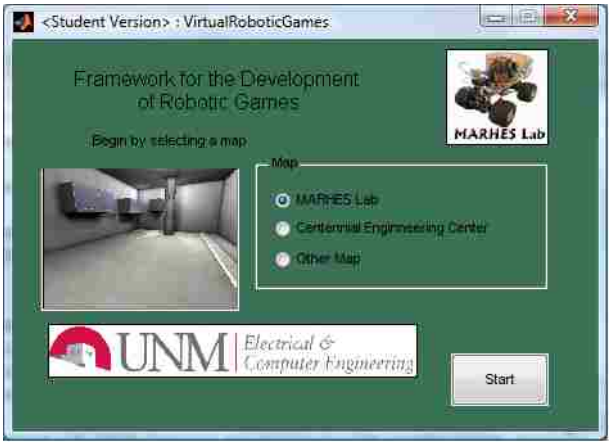

Figure 4.1: The Virtual Robotic Games framework welcome splash.

The new window, shown in Figure 4.2, provided by the framework prompts the user to select a game, determine the number of player robots, their positions, poses, and roles. The user selects "Agent Test" and provides the information for the agents.

The head-up display in Figure 4.3, demonstrates the GUI system. The Reset button places the player into the environment. In the USARSim spectator window, you can see the player's camera view in the upper left corner as well as the user's spectator camera overviewing the robot in its environment. The dashboard has four quadrants each displaying live information regarding the player's sensor readings. Two push buttons are provided to move the player. The goto button allows the Robot Device module to use the linear feedback algorithm to send the player to the goal coordinates. The manual button switches the player's controls to the first connected joystick. Using the first axis and the first two buttons, a user can control the yaw and forward velocity of the player.

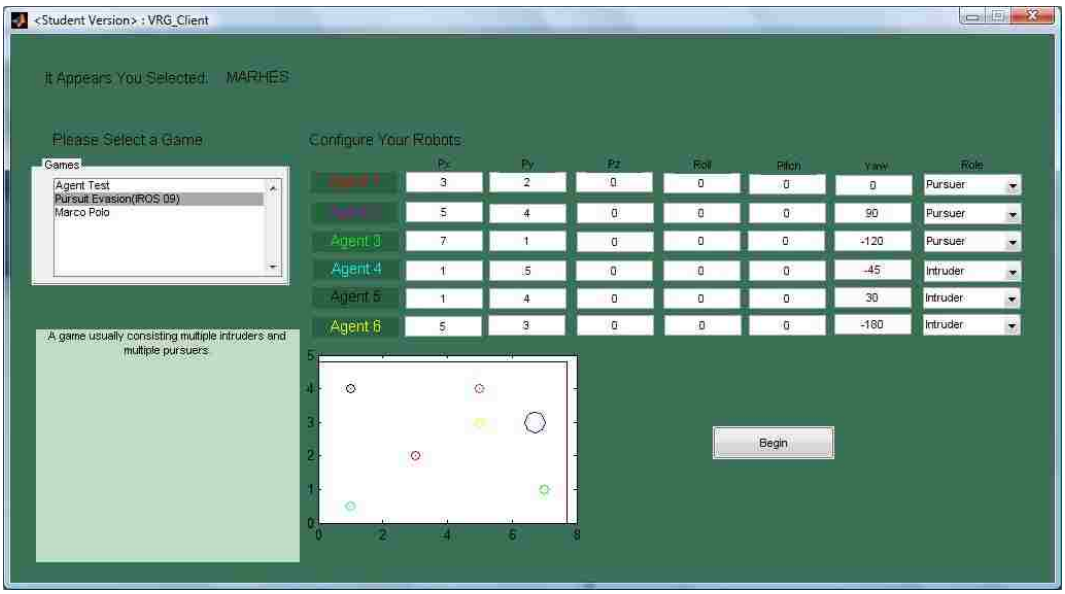

Figure 4.2: The Virtual Robotic Games Framework Client Configurator.

Figure 4.3: The Player HUD game in the virtualized ECE L217 MARHES lab.

Figure 4.4: The Lion and Man problem.

4.2 Robot Pursuit-evasion Game.

A pursuit-evasion game has several distinct features. Similar to the Lion and Man problem derived by Rado[44], a set of P pursuers and a set of I evaders are placed in the same environment $O \subset \mathbb{R}^2$. Both pursuers and evaders have the same maximum forward velocity and the goal for the team of pursuers is to choose values of $u_p = [v_p \ \omega_p]$ for all pursuers to pursue and capture every evader. The team of evaders must choose their values of $u_i = [v_i \ \omega_i]$ for each evader to evade the pursuers. The pursuit-evasion game demonstrates detection and capture algorithms, path estimation, and interception prediction.

4.2.1 Game Description

This game is a simple intruder detection and capture. In this game, the scenario will consist of three pursuers and two evaders. The evaders are called Intruders and will intrude a certain convex boundary, called the Surveillance Area in Figure 4.5, guarded by the Pursuers. The Intruders will start outside of the Surveillance Area and at random time intervals, begin by entering into the Surveillance Area.

The game constrains the Intruder's movements to a linear path to allow the Pursuers some Intruder predisposition. This predisposition allows the Pursuer to project a guaran-

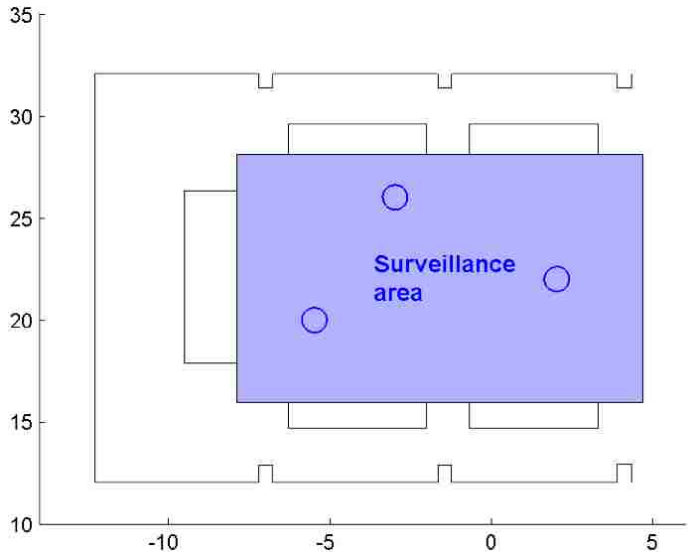

Figure 4.5: A Surveillance Area in the Centennial Engineering Center map.

teed interception point for each Intruder's linear trajectory. After two Intruder sightings, the Pursuers must determine a capture point and dispatch a Pursuer to capture the Intruder. This application is realisitic when considering piece-wise linear trajectories. When the dispatched Pursuer and targeted Intruder are within a certain range, the Game Coordinator counts this as a point for the Pursuers and sends a captured signal to the Intruder. If the Intruder is outside of the Game Coordinator's Surveillance Area boundaries, the Intruders receive a point and the Intruder is halted and the Pursuers are notified. The game is finished when all of the Intruders have been captured or have escaped through the Surveillance Area.

4.2.2 Game Coordinator

Algorithm 4.1 The Pursuit-evasion Game Coordinator Algorithm

```
Assign random launch time for each Intruder
Perform initial robot placement
while Game_Status = True do
    if Timer > IntruderTime then
        Send Intruder.Go
    end if
    while NumberDetections < 2 do
        Poll Pursuers for Detections
    end while
    Receive Pursuer Interception Distances
    Send Dispatch_OK →Pursuer
    while Pursuer_Dispatched = True do
        if Distance between Intruder and Pursuer <1 Meter then
            Pursuer_Score = Pursuer_Score + 1
            Send Reset to All Players and Dispatch_OK
        elseIntruder Outside of Surveillance Area
            Intruder_Score = Intruder_Score + 1
            Send Reset to All Players and Dispatch_OK
        end if
    end while
    if Last Intruder or Timer Ends then
        Game_Status ← False
    end if
end while
```

The Game Coordinator has several key roles. It initiates the players and for each Intruder assigns a start time, with one exception who is always assigned to zero seconds. As the first Intruder begins, it ensures that two detections are received before Pursuers may dispatch. Then it makes the decision whether an Intruder has been captured or was able to escape. The Game Coordinator ends the game when the game timer runs out or when all the Intruders are accounted for. Its algorithm is shown in Algorithm 4.1.

4.2.3 Intruder

The Intruder object is simple. Due to the Intruder's kinematic constraints its kinematic model simplifies to Equation 4.1, where $v_i = v_{max}$ is its constrained velocity and $\theta_i = c$ is it's constrained bearing, where $c \in [-\pi, \pi]$. Its behavior is depicted in Figure 4.6.

$$\begin{aligned} \dot{x}_i &= v_i \cos\theta_i, \\ \dot{y}_i &= v_i \sin\theta_i, \end{aligned} \tag{4.1}$$

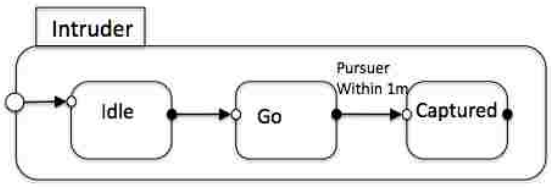

Figure 4.6: The Finite State Machine of the Intruder Behavior.

4.2.4 Pursuer

The Pursuer is the player. It's goal is to collaborate with multiple laser readings to track and capture the Intruder. Using the SICK Laser on each Pursuer, an environment snapshot is taken to determine static wall artifacts. Intruder sightings are recorded as unusual readings and Intruder sightings are accumulated such as in Equation 4.2, where $\mathbf{Y} = [y_1 \dots y_i \dots]^T$. Intruder trajectory tracking is computed using a Least Squares Linear Fit, Equation 4.3, to determine a linear model. This is accomplished using MATLAB's polyfit to return a first degree polynomial fit. Velocity is known to be constant and is computed from two point readings.

$$\mathbf{Y} = \mathbf{X} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_i & 1 \\ \vdots & \vdots \end{bmatrix}, \tag{4.2}$$

$$\begin{bmatrix} a \\ b \end{bmatrix} = \left(\mathbf{X}^T \mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{Y}. \tag{4.3}$$

When an Intruder trajectory has been estimated, the pursuers compute Intruder interception points, Figure 4.7. Intruder interception points are estimated by the Equation 4.4, where k is the Pursuer gain coefficient. This means the Pursuer dispatched will reach the interception point $1/k$ times faster than the target Intruder.

$$\frac{d_p}{\bar{v_p}} = k\frac{d_i}{\bar{v}_i},\ 0 < k \leq 1 \tag{4.4}$$

After the interception points have been determined, the Pursuers send their d_i's to determine who will be dispatched. The marketplace is maintained by the Game Coordinator

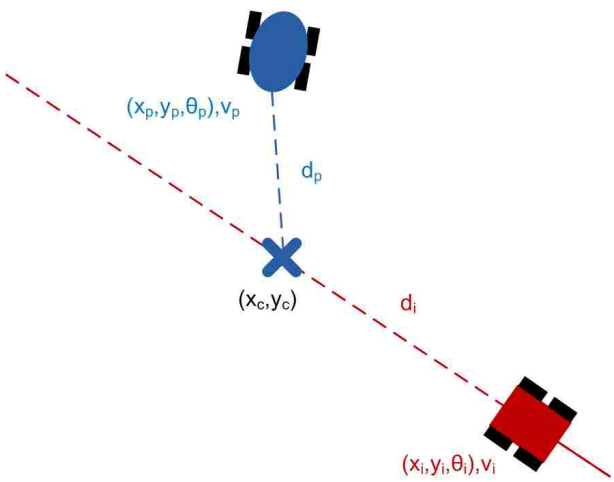

Figure 4.7: Intruder Interception Point.

and the closest interception point to the Intruder is selected and the Pursuer is dispatched. Modeled by the skid-steered kinematic model from Equation 2.2 and using the control law described in Figure 2.4, the dispatched Pursuer goes to their Intruder interception point to intercept and capture the Intruder.

4.2.5 Experiment Results

The simulation is set up with the initial conditions given in Table 4.1. The Pursuers are represented by the three Pioneer2-AT robots in Figure 4.8, and the Intruders are the two iRobot ATRV-Jr robots. The simulation is placed in the courtyard of the Centennial Engineering Center at the University of New Mexico.

The simulation begins with the same introduction screens as the Player HUD experiment. The Game Coordinator prompts the user to choose the environment, number of robots, their roles, and their initial position and orientation. Soon after the user prompts the Game Coordinator to begin, the players are placed in their defined starting positions on the map and the Intruders are set to advance into the Surveillance Area. The Pursuers

Table 4.1: The Pursuit Evasion Initial Conditions.

Name	X	Y	Yaw
Pursuer 1	-7.00	-20.00	0.00°
Pursuer 2	3.00	-22.00	180.00°
Pursuer 3	-3.00	-26.00	0.00°
Intruder 1	-7.50	-16.00	−60.00°
Intruder 2	-1.30	-28.40	105.00°

Figure 4.8: The Pursuit Evasion initial experiment setup.

are equipped with SICK Laser Scanner LMS200 to detect the Intruders.

After two Intruder sightings are acquired, the Pursuers determine their Interception Points and place their costs to reach the point. The Game Coordinator returns the lowest cost to each Pursuer. Comparing their original cost to the returned cost, the Pursuer with the lowest cost can confirm it is the closest to the Intruder and begin its pursuit. The data for this simulation was plotted in Figure 4.9 and Figure 4.10 shows the two Intruders successfully captured.

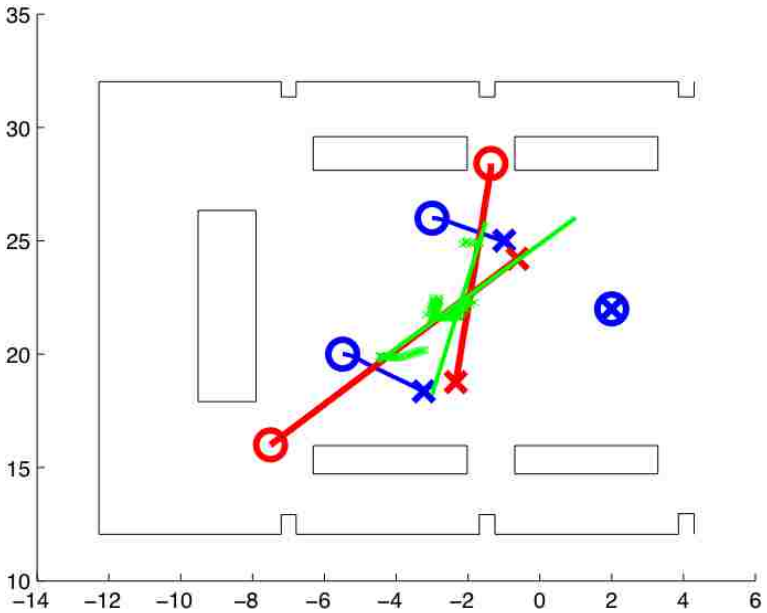

Figure 4.9: The Pursuit Evasion Simulation Data.

Figure 4.10: The Pursuit Evasion Conclusion.

4.3 Sharks and Minnows

Sharks and Minnows was originally based on the swimming game. The game is set up to have two teams: Sharks and Minnows. In a rectangular field $O \subset \mathbb{R}^2$ with dimension $L\ x\ W$, a set of P players are initialized as Sharks and Minnows. Sharks are initialized on one side of the field and Minnows on the opposite side. During the game, players who are Sharks are required to tag Minnows while players who are Minnows are required to traverse across the playing boundary and avoid the Sharks. If a Minnow is tagged by a Shark, that Minnow becomes a Shark after the round is over. Depending on the initial number of Minnows and Sharks, a limited number of Minnow passes are allowed and after the last Minnow pass, any remaining Minnows win the game, otherwise the original Sharks win.

Figure 4.11: The Shark and Minnows game.

4.3.1 Game Description

The game is played on a medium sized rectangular field . Generally the field must be large enough to accommodate the players to move freely. The number of Minnow passes are determined by Equation 4.5. A game of six Minnows and one seed Shark, would have two Minnow Passes. Each Minnow Pass begins with the Sharks lined up on one side and the

Minnows on the other. They will make a pass across the field lengthwise. Any Minnows who are caught within D_{tag} distance from a Shark are considered tagged. Both Minnows and Sharks are free to move any direction as long as they are within the boundary O.

$$Minnow\_Passes = \left\lfloor \frac{Minnows - Sharks}{2} \right\rfloor \tag{4.5}$$

Tagged Minnows switch roles and begin to help Sharks. This changes the dynamics of the game, because the Minnows quickly lose the advantage as their numbers decrease in a geometric progression. Therefore, it is crucial for Minnows to minimize their losses early in the game while vise versa the Sharks need to increase their numbers early. The players have full awareness of the environment and where the sharks are. Differences in each player is introduced by giving the players a random maximum forward velocity. This creates an interesting effect as some minnows become stragglers and can be singled out from the crowd.

4.3.2 Game Coordinator

Due to the nature of Sharks and Minnows, the Game Coordinator has a more defined role as a referee. It does not intervene with the player's decision tree and only prompts the players when they are either tagged, step out of bounds, or beginning a new Minnows pass. Its algorithm is described in Algorithm 4.2. When initialzing each player, the Game Coordinator will determine that specific robot's maximum velocity.

Algorithm 4.2 The Sharks and Minnows Game Coordinator Algorithm

```
Initialize Playing Field and Players
while Game_Status = True do
    Send Round_Start to players
    if Shark and Minnow within 30 cm then
        if Minnow is in End Zone then
            Do Nothing
        else
            Set Minnow to Inactive
        end if
    end if
    if Player is out of bounds then
        Set Player to Inactive
    end if
    if All Remaining Minnows are in End Zone then
        Set all Inactive Players to Shark
    end if
    if No more Minnows then
        Sharks Win
    else if No more Minnow Passes then
        Minnows Win
    end if
end while
```

4.3.3 Player

The Players are have three distinct states described in Figure 4.12. Minnows begin in the *Minnows* state, Sharks are in the *Sharks* state, and any player who was tagged or ran out

of bounds are set to *Inactive*. The *Minnows* state is the evader, it must get to the other side without being tagged by any of the *Sharks*. The *Minnow* strategy consists of a scalar vector field F over the O space. This field is called a potential field and it produces a force $-\nabla F$ on the Player. The field is linearly toward the safe end zone and linearly away from the barycenter of the *Sharks*, Figure 4.13. A more immediate algorithm is exhibited if a *Shark* is known to be closer than two meters, then the *Minnow* will directly evade that specific *Shark* without going out of bounds.

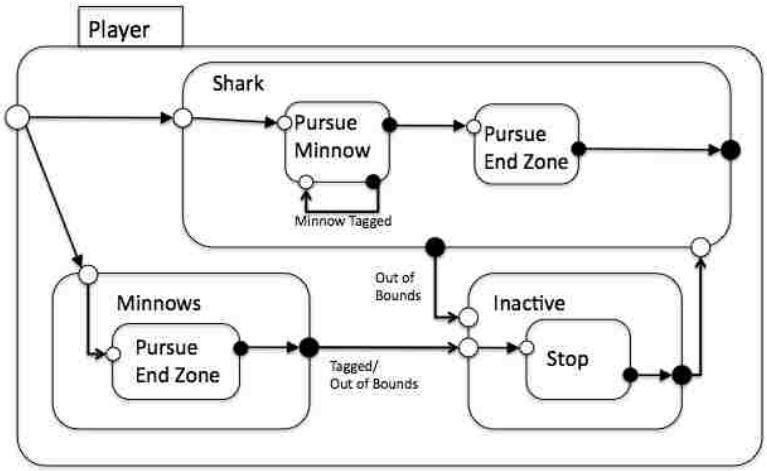

Figure 4.12: The Player Finite State Machine.

When a *Minnow* is tagged by a *Shark*, the Game Coordinator will set that Player to *Inactive*. This will also occur when a *Minnow* is out of bounds. When a Player is *Inactive*, it is forced to stop and remain in its current position. The *Inactive* Player will not resume movement until the Game Coordinator assigns it as a *Shark*. *Sharks* are the pursuers, they are decentralized and contains the same algorithms as the Pursuers in the Pursuit-Evasion experiment. *Minnow* tracking will not be required as there is no shroud in this experiment. The *Sharks* will approach the nearest *Minnow* and attempt to intercept as many as possible during a Minnow pass.

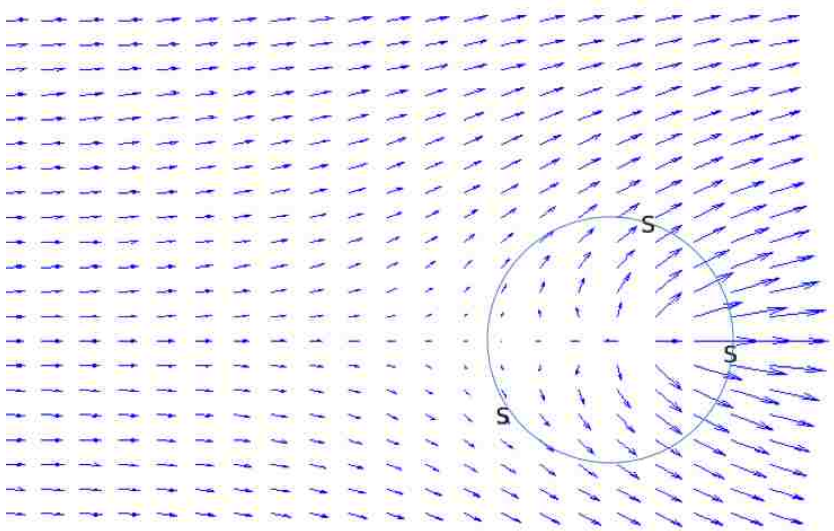

Figure 4.13: The Minnows Potential Function.

4.3.4 User Interface

The Game Coordinator will allow the user to resume the role of one of the players in the game. The control schemes are from the Player HUD experiment. The user can select which player to control but will not know the maximum forward velocity.

4.3.5 Experiment Results

The Sharks and Minnows experiment was held on the opposite side of the Centennial Engineering Center courtyard. The initial conditions are given in Table 4.2 and can be seen in Figure 4.14

The experiment had two Minnow Passes. The *Shark* targeted the quickest *Minnow* first and was able to tag it, shown in Figure 4.15. It recalculated a new *Minnow* and was able

Table 4.2: The Sharks and Minnows Initial Conditions.

Name	X	Y	Yaw	Role
Player 1	21.00	-30.00	90.00°	Shark
Player 2	17.00	-16.00	−90.00°	Minnow
Player 3	19.00	-16.00	−90.00°	Minnow
Player 4	21.00	-15.40	−90.00°	Minnow
Player 5	21.50	-16.00	−90.00°	Minnow
Player 6	22.00	-15.00	−90.00°	Minnow
Player 7	24.00	-15.00	−90.00°	Minnow

to tag it as well due to its superiority in velocity. In Figure 4.16, the Minnow pass ended and the first *Minnow* to be tagged remained in place since it was rendered *Inactive*. The Minnow pass ended with four remaining *Minnows* and three *Sharks*.

The experiment ended with the *Sharks* winning. After multiple experiments, the *Sharks* appear to have the advantage in this game and a new strategy may need to be developed to exploit weakness to guarantee that the *Minnows* win.

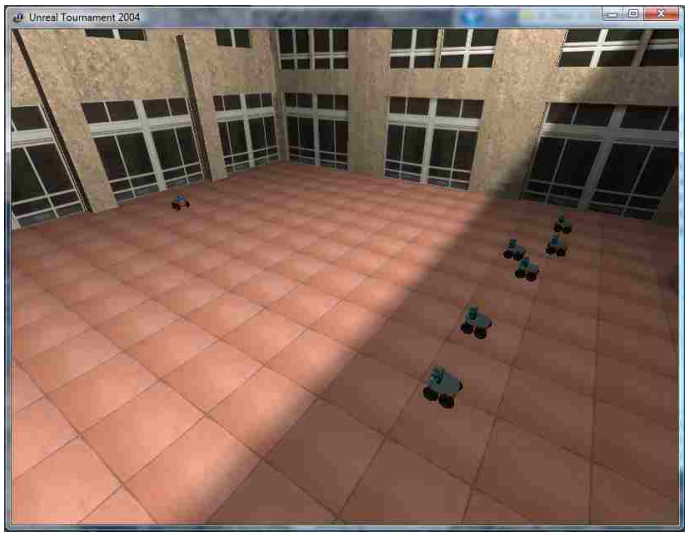

Figure 4.14: The Sharks and Minnows initialization.

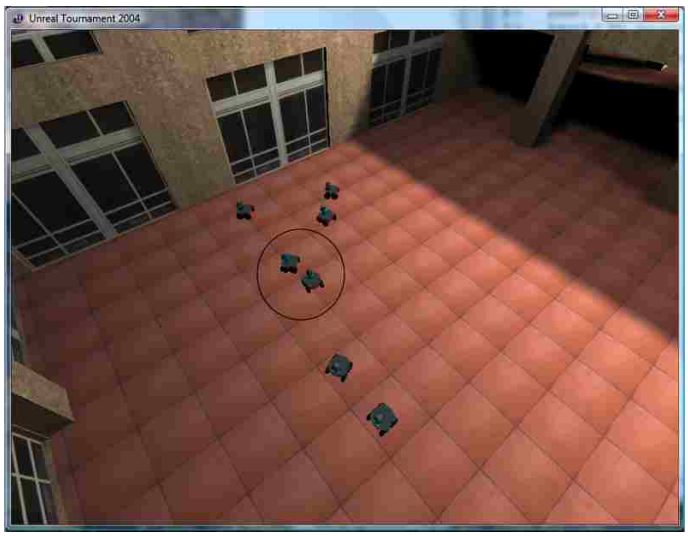

Figure 4.15: The Sharks and Minnows tagging situation.

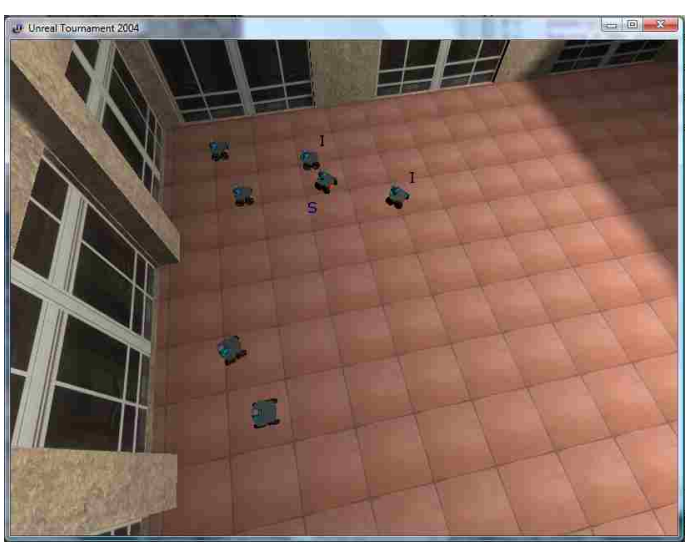

Figure 4.16: The Sharks and Minnows first Minnow Pass's results.

4.4 Roboflag

RoboFlag is a testbed developed at Cornell University in 2001 [25]. Modeled after "Capture the Flag", the rules and strategies have remarkable resemblance. Two teams compete against each other and the goal of the game is to receive the most amount of points. This game is implemented into the framework through three key roles. The Rules and Interactions are governed by the Game Coordinator module, the Player's pathplanning, decision making, and cost bids are described in the Robot Device module, and the user input and scoreboard are managed by the User Interface module. These three roles provide a comprehensive demonstration for the Virtual Robotic Games framework.

4.4.1 Game Description

The game is set on a playing field that is six meters long and four meters wide as depicted in Figure 4.17. The field is split in two equal sides: a red side and a blue side. On each side, the team's home zone and flag are fixed and their locations are priori information in each player. The objective for each team is to invade the other side, enter the other team's flag zone and take their flag. Simultaneously, the other team is defending their flag and can tag the invader, forcing the invader to directly return to its home zone. This mix of offense and defense provides a rich game scenario for developing strategies.

Each team's playing half denotes the boundary where the players can tag opponents. The home zone is a quarter circle in a corner of the playing side. Its purpose is to provide a destination for tagged players to return and is the flag return point for players with the opponent's flag. The flag zone is a full circle where players defend their flag which is in the middle of the circle. Players within the flag zone cannot be tagged and defenders are posted outside of this region.

The interaction rules correlate to the different zones. A player becomes *Tagged* if he's

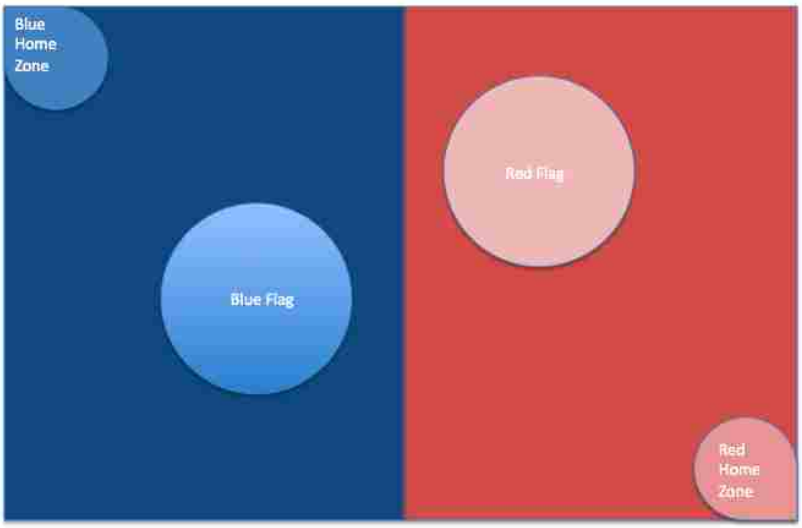

Figure 4.17: The RoboFlag playing field.

in *Normal* operation, on the opponent's side of the playing field, and comes within 30 centimeters of an opponent player. He then must return to his home zone to untag himself. A player becomes *Flagged* if he's within 30 centimeters of the opponent's flag position. He then must return to his home zone to score for his team. If a *Flagged* player comes within 30 centimeters of any opponent anywhere in the field, he becomes *Tagged* and the flag returns to its place in the opponent's flag zone. The player then follows the rules as a *Tagged* player and must return to his home zone to untag himself. When any player is within 60 centimeters to its home zone corner, he becomes *Normal*.

Points are tallied as flag captures. The first team with three flag captures wins. A RoboFlag game is twenty minutes with two rounds at ten minute intervals. Tied games continue to a final sudden death where the first flag captured wins the game. This tiebreaker round is allotted five minutes and the game is considered a draw if no flags are captured during this round.

The purpose of the RoboFlag competition is to provide a benchmark for human machine interaction. Each team has a human operator who does not send drive commands

but rather communicates strategies to his team players. Like a coach, the role is not to be cumbered by low-level robotic logic but rather simple and efficient by sending high level strategies. This human machine interaction sets this experiment apart as it is a playable game rather than an experiment to sit back and observe.

4.4.2 Game Coordinator

The Game Coordinator's role is to maintain gameplay, scorekeeping, timekeeping, and to ensure rules are kept for each player's interactions. Its algorithm is described in Algorithm 4.3. The Game Coordinator is responsible for initializing the playing field and players. After the initial placement is set, it sets each player to *Normal* and the game begins. It checks interactions and the game timer and acts accordingly. The Game Coordinator is the referee and communication protocol.

Other than proctoring the game, the Game Coordinator is also called when a player is *Tagged*. A *Tagged* player forfeits control and is then managed by the Game Coordinator. It guides the player back to his home zone and control is returned to the player.

4.4.3 Player

The player has three states:*Normal*, $Flagged$, and *Tagged*. A finite state machine depicts these three states in Figure 4.18.

The *Normal* state is the default player behavior when it is not tagged nor returning the flag. It consists of a simple logic loop that allows developers to input their robotic code and apply their strategies to the game. Due to the necessity for offenses and defenses, optimization may be necessary to determine a good blend of strategies.

Algorithm 4.3 The RoboFlag Game Coordinator Algorithm

```
Initialize RoboFlag Playing Field and Players
while Game_Status = True do
    Setup Marketplace
    Send previous lowest bid to Normal Players
    Record and Update lowest bid
    if Players within 30 cm then
        if Between Player and its Home Zone then
            if Player is Flagged then
                Add one to Player's Team Score
            end if
            Set Player to Normal
        else if Between Player and Opponent's Flag then
            if Player is Normal then
                Set Player to Flagged
            end if
        else if Between Player and Opponent then
            if Blue Player in Red Side then
                Blue Player is Tagged
            else if Red Player in Blue Side then
                Red Player is Tagged
            end if
        end if
    end if
    if Time is Up then
        Determine Gameplay
    end if
end while
```

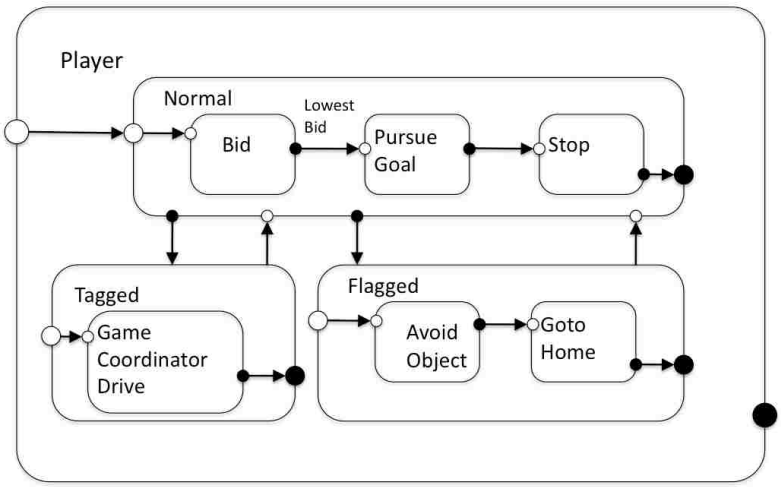

Figure 4.18: The RoboFlag Player Finite State Machine.

A conservative strategy is to dispatch one player to collect the opponent's flag. The players place bids among each other to determine who is closest to the opponent's flag. This can be construed as energy costs and the lowest bid would be the most efficient. The remaining players become defensive and guard their flag. This strategy benefits from the greater defensive stance, however, may prevent the team from gaining opportunity to score.

An offensive strategy is to dispatch all players to collect the opponent's flag. The players then create a formation to shield the opponents from the flag carrier. Two formations can be produced, players can separate into two groups to divide the opposing players or as one group ensuring players remain as obstacles after they are tagged to protect the flag carrier.

The *Flagged* state is executed when a player obtains their opponent's flag. The algorithm is a fast evasion code with its priority is evading opponents and an ultimate goal of reaching its home zone to score. Some alterations are allowed to create a smarter flag return. These include avoiding opponents, cooperating with teammates to intercept in-

coming opponents and calculating safer routes.

The *Tagged* state is active when a player interacts with an opponent within the opponent's playing side or interacts with an opponent while holding the opponent's flag. A *Tagged* player is controlled by the Game Coordinator and the Robot Device becomes transparent, directly submitting its position and orientation and receiving wheel speed directly from the Game Coordinator.

4.4.4 User Interface

The User Interface used for RoboFlag is specialized to a higher-level command center. Information regarding the game is displayed in a GUI while commands are sent through the gamepad. The user is capable of both low-level and high-level control. When clicking on one of the four player control buttons, the control layout on the gamepad is the same as the previous experiments. Each of the player control buttons designate a specific robot player.

When none of the player control buttons are pressed, the GUI is set up as a commander style interface. Each of the four face buttons represent a certain priority. These four priorities are: $Team\ Flag$, $Tag\ Defense$, $Flagged\ Safety$, and $Hail\ Mary$. $Team\ Flag$ means the priority is safeguarding the team's flag. Generally all players will swarm around the team's flag or attempt to directly tag the Opponent's $Flagged$. The $Tag\ Defense$ priority represents the priority to quickly tag any opponents in the team's playing side to quickly purge the field. $Flagged\ Safety$ is the strategy to sacrifice oneself to shield its $Flagged$ from being tagged. $Hail\ Mary$ is similar to most games with $Hail\ Mary$'s. All players attempt to capture the opponents flag as an aggressive stance to make a score. The RoboFlag User Interface is an interesting problem that introduces a human machine interaction that could possibly improve the way human and machines communicate with each other.

4.4.5 Experiment Results

This experiment required a larger field due to the size of the Pioneer2-AT. The playing field was doubled in size from the Cornell University specifications. The players were lined up to begin play as in Figure 4.19. When the Game Coordinator began the round, the players placed their bids on who will attempt to capture the opponents flag. Using the consensus theory described previously, the players computed and compared who's bid is lowest and that player began their trajectory toward the opponent's flag.

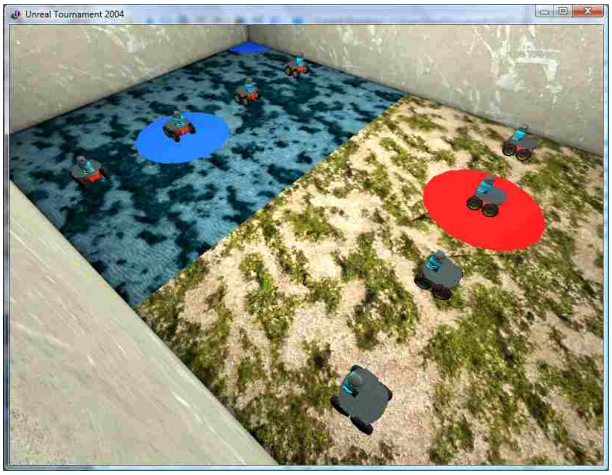

Figure 4.19: The RoboFlag Virtual Field.

Set to instigate results, the blue team was limited on using the conservative strategy and the red team was limited to only the aggresive strategy. This ensured both strategies were the Achilles' heel to each other. The Blue team would be out numbered when all four red players entered the field and the Red team would not have any defenders to tag the single blue flag taker. In conclusion, the blue team ensured victory because the entire Red team was incapacitated due to all its players were *Tagged*.

When the User Interface is implemented, the user plays the game as the commander of his team's players. The user's opponent's strategy has been set on a balanced approach, sending only one player for the flag and leaving the rest for defense. The user's two

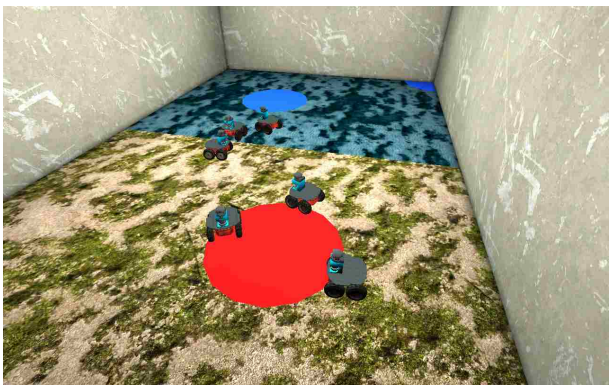

Figure 4.20: Roboflag when User commands Defense.

major priorities are *Defense* in Figure 4.20 and *Hail Mary* Figure 4.21. A successful user strategy is to do full defense until at least two of the opposing players are tagged and returning to their home base. Then while two of the opponent's players are *Tagged*, the user commands *Hail Mary* and sends his entire team to grab the flag. The players who are tagged first, become *Tagged* and shields the rest of the players from the opposing players. When one player captures the flag, the user commands *Defense* to convoy the flag carrier back to home base.

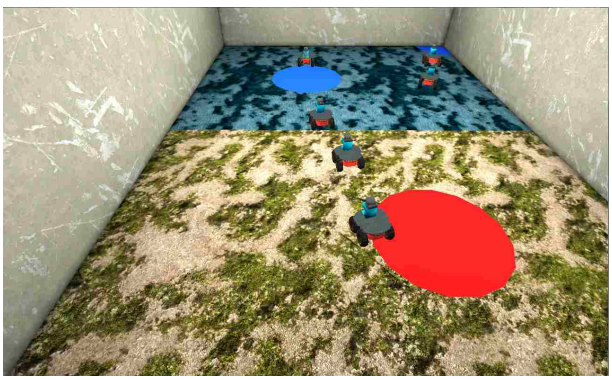

Figure 4.21: Roboflag when User commands Hail Mary.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, a framework for the development of Virtual Robotic Games has been presented. The framework has been implemented on a cross-platform software environment to present a flexible research tool for developing robotic scenarios and their robot's tasks. It brings accurate dynamics, realistic three-dimensional rendering, vast robot and sensor models in efficient real-time simulations. There are many robot simulators, however, The Virtual Robotic Games framework integrates a role-based robotic environment that allows developers to focus directly on one role module without the need to update the entire framework.

The framework compartmentalize Robotic Games' agents by their different roles. The Game Coordinator module maintains the game itself, checking conditional rules and ensuring the gameplay remains bound to their conditions. The Robot Device module is a robot abstraction layer, maintaining robotic algorithms and hardware middleware. The User Interface module is a dedicated module to develop a graphical display and gamepad input experience that captures playing the Robotic Game. Separating the modules and

building their relationships have defined the framework's architecture, methodology, and modularity.

Several Robotic Games have been evaluated to demonstrate the framework. Robot Pursuit Evasion, Sharks and Minnows, and Robotic Capture the Flag have all demonstrated their own unique characteristics. Pursuit Evasion has tested and proven theoretical guarantees to track and capture Intruders in a multiple point of view system. Sharks and Minnows has demonstrated an extension of pursuit-evasion by adding the dynamics of the swimming pool classic. Using repeated match data, Sharks and Minnows strategy can be further improved to extend development in robotic algorithms. The RoboFlag experiment provided the framework for developing human machine interfaces, introducing high-level communication, where robots will study the semantics of command before acting.

The Virtual Robotic Games framework provides a cost-effective, role-based behavior control environment that promotes collaboration, education, and research in robotics. It brings a development platform that is focused on the development of Robotic Games and Robotic Games strategies. The Virtual Robotic Games framework has provided a comprehensive test bench that successfully simulates these games and provided a research platform for robotic behavior.

5.2 Future Work

Further development to expand the framework can incorporate physical robot limitations. Battery resources can be considered by a total energy function [45]. Noisy or limited sensor readings can be solved by a fusion center [46] and limited or noisy communication channels should be modeled in USARSim. USARSim provides aerial vehicles which can be added to the Robotic Games for more intriguing gameplay.

Epic Games has released the next physics engine with Unreal Tournament 3 (UT3).

UT3 is backed by the third generation Unreal Engine 3 (UE3), which supports more realistic render techniques such as dynamic shadows, high dynamic range rendering, per-pixel lighting, global illumination, volumetric environmental effects, and an extensive particle system. UE3 also advanced their physics system, powering it by the next generation NVIDIA's PhysX. Rigid body physics improves realism between objects, ragdoll characters, and other complex dismemberable objects.

USARSim has already begun developing their modification to UT3. The framework can be adapted to run on the UT3 USARSim system by rebuilding the MATLAB USARSim toolbox to work with UT3's network protocol. Additionally screen capture would have to be implemented to receive camera video and a new image server will have to be developed. A new image server and MATLAB toolbox would allow the framework to utilize UT3's physics system, thus, advancing our simulations with video realism and more accurate physic dynamics.

Currently the framework does not have an outlet to advance its simulations to hardware. Several hardware abstractions are available with potential to be integrated into the Virtual Robotics Games framework. The Player device server [47] is an open-source framework that is validated [33] in the academic community for support of a large variety of robotic hardware and robot platforms. Another software framework is the Microsoft Robotics Developer Studio [48]. This platform supports several standard robot platforms, however, integration with the Virtual Robotic Games framework may not be cost effective. A hardware abstraction platform integrated with the framework will expand development to hardware realization, allowing users to further validate their code on physical platforms.

In Chapter 4, many robotic games and scenarios have been defined and preliminary player algorithms have been developed to complete the tasks required. Theoretical guarantees of performance should be investigated. Research by the robotics and pursuit-evasion communities may address the complexity and stability of applicable coordination algorithms. Incorporating all work will develop a more complete definition and solution for

the study of Robotic Games.

Appendix A

Software Installation

The Virtual Robotic Games Framework is comprised of two software packages:

- MATLAB
- USARSim

A.1 MATLAB Installation

1. Install MATLAB as recommended by Mathworks. Ensure two packages are downloaded and installed.
2. MATLAB packages
    - Simulink 3D Animation 5.1.1
    - MATLAB USARSim Toolbox Beta Version 1.1 [42]

      The java code provided in MATLAB USARSim Toolbox is specifically compiled for Windows. For Mac OS X and Linux java must be compiled with a JAVA Compiler to ensure binary compatibility

3. Place MATLAB USARSim Toolbox inside VirtualRoboticGames folder. Enter Folder. For quick start, Virtual Games Framework runs as a gui from the prompt line. Run as

```
EDU>> VirtualRoboticGames
```

This should open Figure 4.1.

A.2 USARSim Installation

USARSim's installation requires modifying a commercial product and is more involved.

1. To install USARSim, you must have a working copy of Unreal Tournament 2004 (UT2004). Install UT2004 as recommended by Epic Games and patch to v3369.
2. Download the USARSim 2004 files from sourceforge [49]. Either install the windows executable or unpack the compressed archive file into the UT2004 directory.
3. For Mac OS X and Linux, Install Wine
4. Compile USARSim by running make.bat or running wine in the Systems folder

```
$ wine ucc.exe make -ini=USARSim.ini
```

5. Run USARSim Server from the Systems folder

```
$ ./ut2004/System/ucc-bin-linux-amd64 server
[map-name]?game=USARBot.USARDeathMatch?TimeLimit=0?GameStats=False
-ini=USARSim.ini -log=usar_server.log
```

Appendix B

Virtual Robotic Games Protocols

B.1 MATLAB USARSim Toolbox

The VRG framework follows the MATLAB USARSim Toolbox. Using the toolbox's protocols with Gamebots/Unreal, VRG sits nicely on top benefiting from the abstraction from Unreal's technical protocol.

Several Commands are:

- Initialize a robot:

```
robot = initializeRobot{robot_name, robot_class,
robot_position, robot_orientation}
```

Where:

robot_name	*robot_name* is the name of the robot.
robot_class	USARSim's robot model class. VRG uses P2AT.
robot_position	*robot_position* is a tuple of the starting position in meters.
robot_orientation	*robot_orientation* is a tuple of the starting orientation in radians.
robot	*robot* is the robot object created.

- Send Drive Commands

```
sendDriveCommand{robot, robot_command}
```

Where:

robot	*robot* is the robot object.
robot_command	*robot_command* = [left_motor_speed, right_motor_speed]

- Get Vehicle State:

```
{State} = getVehicleState{robot}
```

Where:

robot	*robot* is the robot object.
State.TimeStamp	*TimeStamp* is the number of seconds USARSim.
State.FrontSteer	*FrontSteer* is the steering angle of the front wheels of an ackermann steered robot.
State.RearSteer	*RearSteer* is the steering angle of the rear wheels.
State.LightToggle	*LightToggle* is a boolean value for the robot's headlights.
State.LightIntensity	*LightIntensity* is currently not implemented.
State.BatteryLife	*BatteryLife* is a value from 0 to 100, 100 represents full charge.

- Get Vehicle Geometry:

```
{Geo} = getVehicleGeometry{robot}
```

Where:

robot	*robot* is the robot object.
Geo.Name	*Name* is the name of the robot.
Geo.Dimensions	*Dimension* is currently not implemented.
Geo.CenterOfGravity	*CenterOfGravity* is currently not implemented.
Geo.WheelRadius	*WheelRadius* is the robot's driving wheel radius in mm.
Geo.WheelSeparation	*WheelSeparation* is the robot's track width.

- Get INS Readings:

```
{Ins} = getINSReadings{robot}
```

Where:

robot	*robot* is the robot object.
Ins.Name	*Name* is the name of the robot.
Ins.Position	*Position* is a tuple containing the robot's Cartesian position.
Ins.Orientation	*Orientation* is a tuple containing the robot's pose.

- Get Encoder Readings:

```
{Encoders} = getEncoderReadings{robot}
```

Where:

robot	*robot* is the robot object.
Encoders.Name	*Name* is the name of the robot.
Encoders.Ticks	*Position* is a 1x n vector of the ticks recorded from n encoders.

- Get Touch Sensor Readings:

```
{Touch} = getTouchSensorReadings{robot}
```

Where:

robot	*robot* is the robot object.
Touch.Name	*Name* is the name of the robot.
Touch.ContactState	*ContactStat* is a 1x *n* vector of the boolean state recorded from *n* touch bumpers.

- Get Laser Sensor Readings:

```
{Laser} = getLaserSensorReadings{robot}
```

Where:

robot	*robot* is the robot object.
Laser.Name	*Name* is the name of the robot.
Laser.Resolution	*Resolution* the resolution of the laser range finder.
Laser.FOV	*FOV* the Field of View (zoom) value of the laser range finder.
Laser.Scans	*Scans* is a 1x *n* vector of the range detected.

- Get Sonar Sensor Readings:

```
{Sonar} = getSonarReadings{robot}
```

Where:

robot	*robot* is the robot object.
Sonar.Name	*Name* is the name of the robot.
Sonar.Range	*Range* is a vector 1x *n* of the ranges from *n* sonar readings.

- Get Odometry Readings:

```
{Odometry} = getOdometryReadings{robot}
```

Where:

robot	*robot* is the robot object.
Odometry.Name	*Name* is the name of the robot.
Odometry.Pose	*Pose* is the orientation of the robot.

- get GPS Readings:

```
{GPS} = getGPSReadings{robot}
```

Where:

robot	*robot* is the robot object.
GPS.Name	*Name* is the name of the robot.
GPS.Latitude	*Latitude* is the latitude reading from the GPS.
GPS.Longitude	*Longitude* is the longitude reading from the GPS.
GPS.GotFix	*GotFix* is a boolean determined by if one satellite is locked.
GPS.NumOfSatellites	*NumOfSatellites* is the number of satellites locked.

- Shutdown Robot:

```
shudownRobot{robot}
```

Where:

robot	*robot* is the robot object.

References

[1] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, Feb. 2007.

[2] R. Vaughan, "Massively multi-robot simulation in stage," *Swarm Intelligence*, no. 2, pp. 189–208, Aug. 2008.

[3] R. Falconi and C. Melchiorri, "Roboticad: an educational tool for robotics," in *The International Federation of Automatic Control*, Seoul, Korea, July 2008, pp. 9111–9116.

[4] O. Michel, F. Rohrer, and Y. Bourquin, "Rat's life: A cognitive robotics benchmark," in *European Robotics Symposium*, vol. 44. Springer, 2008, pp. 223–232.

[5] D. Riehle and T. Gross, "Role model based framework design and integration," in *Object-Oriented Programming Systems, Languages, and Applications*, 1998, pp. 117–133.

[6] Wikipedia, "Robotics," Online, 2010, http://en.wikipedia.org/wiki/Robotics.

[7] R. P. M. Buehler and M. Raibert, "Robots step outside," in *International Symposium Adaptive Motion of Animals and Machines*, Limenau, Germany, Sept. 2005.

[8] M. Ciocarlie, C. Goldfeder, and P. Allen, "Dimensionality reduction for hand-independent dexterous robotic grasping," in *IEEE/RSJ International Conference on Intelligent Robots and Systems. IROS 2007*, San Diego, CA, Oct. 2007, pp. 3270–3275.

[9] T. Brogardh, "Present and future robot control development - an industrial perspective," *Annual Reviews in Control*, vol. 31, no. 1, pp. 69–79, 2007.

[10] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 3rd ed. Upper Saddle River, NJ 07458: Pearson Prentice Hall, 2005.

[11] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*, 1st ed. Springer-Verlag London Limited, 2009.

[12] H. Khalil, *Nonlinear Systems*, 3rd ed. Prentice Hall, 2001.

[13] A. van der Schaft and H. Schumacher, *An Introduction to Hybrid Dynamical Systems*. Lecture Notes in Control and Information Sciences 251: Springer-Verlag, 2000.

[14] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, , and O. Sokolsky, "Hierarchical hybrid modeling of embedded systems," in *EMSOFT'01*, Oct. 2001.

[15] D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *IEEE Computer*, vol. 30, pp. 31–42, 1997.

[16] R. Fierro, A. Das, J. Spletzer, R. Alur, J. Esposito, Y. Hur, G. Grudic, V. Kumar, I. Lee, J. P. Ostrowski, G. Pappas, J. Southall, and C. J. Taylor, "A framework and architecture for multi-robot coordination," *The International Journal of Robotics Research*, vol. 21, no. 10-11, pp. 977–995, Oct. - Nov. 2002.

[17] M. Zaratti, M. Fratarcangeli, and L. Iocchi, "A 3D Simulator of Multiple Legged Robots based on USARSim," *Robocup 2006: Robot Soccer World Cup X*, vol. LNAI Vol. 4434, no. 1, pp. 13–24, 2007.

[18] P. Harrison, "The micromouse competition - a brief history," Online, February 2010, http://www.micromouseonline.com/book/micromouse-book/history.

[19] R. S. Rainwater, "Robot competitions," Online, March 2010, http://robots.net/rcfaq.html.

[20] N. Karnad and V. Isler, "Lion and man game in the presence of a circular obstacle," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, 2009.

[21] B. Perteet, J. McClintock, and R. Fierro, "A multi-vehicle framework for the development of robotic games: The marco polo case," in *Robotics and Automation, 2007 IEEE International Conference on*, April 2007, pp. 3717–3722.

[22] M. A. Vieira, R. Govindan, and G. S. Sukhatme, "Scalable and practical pursuit-evasion with networked robots," *Intelligent Service Robotics*, vol. 2, no. 4, pp. 247–263, 2009.

[23] A. Efrat, H. H. G.-B. nos, S. G. Kobourov, and L. Palaniappan, "Optimal strategies to track and capture a predictable target," in *Proc. IEEE Int. Conf. Robot. Automat.*, Taipei, Taiwan, Sept. 14-19 2003, pp. 3789–3796.

[24] H. Gonzalez-Banos, D. Hsu, and J. Latombe, "Motion planning: Recent developments," in *Autonomous Mobile Robots: Sensing, Control, Decision Making and Applications*, S. Ge and F. Lewis, Eds. Boca Raton, FL: CRC Press, 2006, ch. 10.

[25] R. D'Andrea and R. Murray, "The RoboFlag Competition," in *Proc. of the American Controls Conference*, June 2003, pp. 650–655.

[26] T. G. McGee and J. K. Hedrick, "Guaranteed strategies to search for mobile evaders in the plane," in *Proc. American Control Conf.*, Minneapolis, MN, June 14-16 2006, pp. 2819–2824.

[27] G. Lakemeyer, E. Sklar, D. G. Sorrenti, and T. Takahashi, *RoboCup 2006: Robot Soccer World Cup X*. Springer-Verlag New York Inc, 2007.

[28] T. Nakashima, M. Takatani, N. Namikawa, H. Ishibuchi, and M. Nii, "Robust evaluation of robocup soccer strategies by using match history," in *IEEE Congress on Evolutionary Computation*, Vancouver, BC, Canada, July 2006, pp. 3270–3275.

[29] J. Bruce, S. Zickler, M. Licitra, and M. Veloso, "Cmdragons: Dynamic passing and strategy on a champion robot soccer team," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, May 2008, pp. 4074–4079.

[30] B. Browning, J. Searock, P. E. Rybski, and M. Veloso, "Turning segways into soccer robots," *Industrial Robot*, vol. 32, no. 2, pp. 149–156, 2005.

[31] J. Craighead, R. Murphy, J. Burke, and B. Goldiez, "A survey of commercial & open source unmanned vehicle simulators," in *IEEE International Conference on Robotics and Automation*, Roma, Italy, April 2007, pp. 852–857.

[32] A. H. Nathan Koenig, "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator," in *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[33] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th international conference on advanced robotics*, 2003, pp. 317–323.

[34] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "USARSim: A Robot Simulator for Research and Education," in *IEEE International Conference on Robotics and Automation*. Los Alamitos: IEEE Computer Society Press, 2007, pp. 1400–1405.

[35] S. Balakirsky, S. Carpin, and M. Lewis, "Robots, games, and research: Success stories in usarsim," in *Workshop Proceedings of the International Conference on Intelligent Robots and Systems*, St. Louis, Missouri, October 2009.

[36] Z. Kira, R. C. Arkin, and T. Collins, "A design process for robot capabilities and missions applied to microautonomous platforms," in *SPIE Conference on Micro- and Nanotechnology Sensors, Systems, and Applications II*, vol. 7692, Orlando, Florida, April 2010.

[37] C. Scrapper, S. Balakirsky, and E. Messina, "MOAST and USARSim - A Combined Framework for the Development and Testing of Autonomous Systems," in *SPIE Defense and Security Symposium*, Orlando, Florida, April 2006, pp. 1400–1405.

[38] D. S. Office, "Darpa synapse," Online, 2010, http://www.darpa.mil/dso/thrusts/bio/biologically/synapse/index.htm.

[39] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "Bridging the gap between simulation and reality in urban search and rescue," in *Robocup 2006: Robot Soccer World Cup X*, vol. 4434, LNAI. Springer, 2007, pp. 1–12.

[40] S. Balakirsky, R. Madhavan, and C. Scrapper, "Development of a virtual manufacturing framework: From end-user performance requirements to robot competitions," in *SPIE Conference on Unmanned Systems Technology X*, vol. 6962. Orlando, Florida: The International Society for Optical Engineering, April 2008, pp. 9111–9116.

[41] Y. Hur and I. Lee, "Distributed Simulation of Multi-Agent Hybrid Systems," *IEEE International Symposium on Object-Oriented Real-time distributed Computing*, April 2002.

[42] M. Hsieh, "Matlab usarsim toolbox," Online, Feb. 2010, http://robotics.mem.drexel.edu/USAR/.

[43] A. Marshal, "Gamebots," Online, Feb. 2010, http://gamebots.sourceforge.net/.

[44] J. E. Littlewood, *Littlewood's Miscellany*. Cambridge University Press, 1986.

[45] C. C. Ooi and C. Schindelhauer, "Minimal energy path planning for wireless robots," *Mobile Networks and Applications*, vol. 14, no. 3, pp. 309–321, June 2009.

[46] T. C. Aysal and K. E. Barner, "Constrained decentralized estimation over noisy channels for sensor networks," in *IEEE Transaction on Signal Processing*, vol. 56, no. 4, April 2008, pp. 1398–1410.

[47] T. Collett, B. MacDonald, and B. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*, 2005.

[48] "Microsoft robotics studio," Online, http://msdn.microsoft.com/en-us/robotics/default.aspx.

[49] S. Balakirsky, "Usarsim sourceforge," Online, Feb. 2010, http://sourceforge.net/projects/usarsim.