Robotic Soccer Strategizing
Simulation Environment

Alan Chuang & Jackie Siu

A thesis submitted in partial fulfillment
of the requirements of the degree of

BACHELOR OF APPLIED SCIENCE

Supervisor: Beno Benhabib

Department of Mechanical and Industrial Engineering
University of Toronto

March 20, 2008
Acknowledgment

We would like to express our sincere gratitude to our supervisor, Professor Beno Benhabib, for his patient guidance, encouragement, and advices throughout the duration of the thesis. With his assistance, the complexity of the subject has been greatly reduced, which, otherwise, would not have been possible to proceed.
Division of Work

Chapter 1 Introduction & Background written by Alan Chuang

Chapter 2 Program Theory & Logic written by Alan Chuang

Chapter 3 Final Program written by Jackie Siu

Chapter 4 Conclusion written by Jackie Siu

Program Architecture collaborative

Simulation Program collaborative
Abstract

Research on robotics and Artificial Intelligence (AI) has been increasing in recent years with particular interest in robotic soccer due to the complexities involved encompass all the difficulties often encountered in this field of research. By focusing research efforts on this “standard” problem, that is, robotic soccer, it is believed that progress will be accelerated dramatically. This thesis will focus on developing a brand new simple 2D simulation program consisting of a server and a client program based on rules and regulations in the RoboCup’s SSL. However, emphasis will be placed on the server program. The server program will set up the simulation environment, that is, the virtual soccer field, with most real-life variables taken into consideration. The client program will contain a few simple gameplay strategies to ensure working of the server program and the simulation program as a whole. The purpose of this simulation program is to provide an efficient tool for researchers to concentrate on higher level problems such as real-time learning and cooperation of the robots in the RoboCup’s SSL, and at the same time, to allow easy testing and tuning of different gameplay strategies.
# Table of Content

1 Introduction & Background ................................................................. 1
  1.1 Motivation ...................................................................................... 1
  1.2 Background .................................................................................. 2
  1.3 Objectives ..................................................................................... 4
  1.4 Introduction to Simulation ............................................................. 5

2 Program Theory & Logic .................................................................. 7
  2.1 Program Architecture .................................................................. 7
  2.2 Module Characteristics ............................................................... 7
    2.2.1 Blackboard ................................................................................ 7
    2.2.2 Controller Unit ......................................................................... 8
    2.2.3 Simulation Environment ......................................................... 8
    2.2.4 Computer Vision System ....................................................... 8
    2.2.5 Team A and B Strategy ............................................................ 9
  2.3 Program Flow .............................................................................. 10
    2.3.1 Program Startup ....................................................................... 10
    2.3.2 Program Operation ................................................................. 11
    2.3.3 Blackboard – The Significance .............................................. 12

3 Final Program .................................................................................. 14
  3.1 Module Functions ......................................................................... 14
    3.1.1 Graphical Simulation ............................................................. 14
    3.1.2 Robot Controllers ................................................................... 16
    3.1.3 Ball Controller ....................................................................... 18
    3.1.4 Ball Rebound Model .............................................................. 20
    3.1.5 Camera Vision System ............................................................ 22
    3.1.6 Blackboard ............................................................................. 24
  3.2 Program Interface ......................................................................... 25
    3.2.1 User Customizability .............................................................. 25

4 Conclusion ...................................................................................... 27
4.1 Final Product ......................................................................................... 27
4.2 Future Improvements............................................................................. 27
Collection of Figures........................................................................................ 29
Collection of Tables ......................................................................................... 35
References ..................................................................................................... 37
Appendix A: Main Program Source Code ......................................................... 38
Appendix B: PHP Script Code ......................................................................... 48
List of Figures

Figure I Sample screenshot of simulation......................................................29
Figure II SSL Field Dimensions (in millimetres)..............................................29
Figure III Program architecture....................................................................30
Figure IV Program flow................................................................................31
Figure V Blackboard significance ..................................................................31
Figure VI Simulation Field Dimensions (in pixels)...........................................32
Figure VII Ball reflection assumption ............................................................32
Figure VIII Explanation for obtaining robot corner coordinates....................33
Figure IX Explanation for obtaining reflection angle .......................................33
Figure X Simulation interface with user customizable variations ......................34
Figure XI Sample frames in simulation..........................................................34
List of Tables

Table 1 Module Read/Write Capabilities.........................................................35
Table 2 Module-to-Module Communication....................................................35
Table 3 Variable names for robot commands.................................................35
Table 4 Variable names for robot commands continued..................................36
Table 5 Variable names for camera vision system coordinates.........................36
1 Introduction & Background

1.1 Motivation

Robotics and Artificial Intelligence (AI) has always been a very popular and extensive area of research and particularly in the past decade as many scientists and researchers are anxious to bring some revolutionary technological breakthroughs to life. A very well-known organization dedicated to this area of research is the RoboCup Federation, which utilizes robotic soccer as the primary research focus on robotics and as the medium for exchange of the most up-to-date knowledge and information on robotics among professionals, researchers, and enthusiasts. Its ultimate goal is to develop a team of humanoid robots that will defeat the human World Cup champion team under the official regulations of FIFA by 2050. It is extremely challenging as the complexity involved requires knowledge and experience from many different fields and subjects. The goal is feasible only if research efforts are brought together to concentrate on robotics and accelerate the progress. As a matter of fact, robotic research has been ongoing for decades, yet little has been achieved in terms of physical AI robots and integration to real-life, both of which have been thought of since early 1900. This slow progress, along with the aim to develop a research and educational tool for multi-agent system and AI, is the motivation behind this thesis. This thesis will focus on robotic soccer with emphasis on simulation environment. It is hoped that a greater understanding of multi-agent system and AI will be obtained through the research and development process.
1.2 Background

This thesis, like many researches on AI and robotics, is based on RoboCup, a very popular robotic soccer competition held annually by RoboCup Federation worldwide. It is an international research and education initiative aimed to promote AI and intelligent robotics research by providing a standard problem for evaluation of various theories, algorithms, and agent architectures. [1] It is believed that progress will be accelerated by focusing research efforts on one standard problem, that is, robotic soccer. The ultimate goal of RoboCup is that:

"By 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.”  [2]

RoboCup is divided into five different leagues: Simulation League, Small Size League (F-180) (hereby known as “SSL”), Middle Size League (F-2000), Standard Platform League (Sony Aibo and etc.), and Humanoid League. All require designing and building of physical robots except the Simulation League. For each league, there are a different set of rules and regulations, which must be closely followed. This paper will have most relevance to the Simulation League and the SSL.

The Simulation League is a computer-based simulation, where two teams of 11 simulated autonomous robots play soccer in a competitive 2D environment. A 3D environment also exists, but all the motions are relatively slow and awkward since humanoid robots are instead simulated which have substantially more variables than a 2D counterpart. The Simulation League requires installation of the RoboCup Soccer Simulator, which contains two main programs: RoboCup Soccer Server and Monitor.
The server program runs the actual simulation and the monitor program displays the simulation on the computer screen. The server program is a real-time system composed of real-life variables and complexities with a refresh rate of 100 ms. The robots acquire necessary information in a user-defined time interval to analyze the environment and respond accordingly. The main advantage of this league is that it allows researchers to focus on higher level problems such as real-time learning and cooperation instead of solving lower level problems such as communication and hardware problems often encountered in modern complex design. This paper will focus on developing a brand new simple 2D simulation program consisting of a server and a client program based on rules and regulations in the SSL, much like the RoboCup Soccer Simulator. The detail of which will be explained in greater detail in later section of this paper. The Simulation League in RoboCup will only be used as a reference during the research and development process.

The SSL is a 5 versus 5 robotic soccer competition with a size limit of 180 mm in diameter and 225 mm in height. The play field is 6100 by 4200 mm (see Figure II). A standard orange golf ball is used as the soccer ball in the competition. The duration of the match is divided into 2 quarters, 15 minutes each, for a total of 30 minutes. It is important to note that this league imposes a number of restrictions, such as kicking speed and dribbling distance, on the designing and building of robots to prevent robots with excessive mechanical advantages. The intent of the competition is to develop teamwork and coordination in robots, not to demonstrate single robot ability. The 2D simulation program to be developed will also take all these into consideration.
1.3 Objectives

The primary focus of this thesis is to create a simulation environment for the Small-Size League (hereby known as 'SSL') in RoboCup. A brand new simple 2D simulation program will be developed along with simple gameplay strategies. The programming software used to develop the simulation program is Adobe Flash® 8 (hereby known as ‘Flash’), previously known as Macromedia Flash 8, specifically using its embedded developer kit language, ActionScript®, in conjunction with Hypertext Preprocessor, commonly known as PHP script. There are two objectives in this thesis which are summarized below.

The first objective is to develop a simple server program to allow testing and tuning of different gameplay strategies. The server program sets all the environment variables and extracts useful information from the client, which ideally has different gameplay strategies setup and executed based on real-time information from the environment, for simulation.

The second objective is to create a number of simple strategies as the client to verify the server program. The strategies developed are intended for verification of the integrity of the server program only, not the focus of this thesis.
1.4 Introduction to Simulation

A simulation is an imitation of a real-life situation using a similar but simpler model that can easily be manipulated to determine real-life effects. It is basically an experiment run as a model of reality. [3] Simulation can generally be divided into two types: physical/manual simulation and computer simulation. This thesis will focus on computer simulation where computer programs based on mathematical models are used. A computer simulation saves time and cost, and more importantly, accelerates the research and development process.

In order to perform a computer simulation, a simulation environment first needs to be created. A simulation environment is a virtual space that duplicates a real-life environment with all possible real-life variables. The simulation environment, in this thesis, is a virtual soccer field, as shown in Figure 1, generated using Flash. However, it is important to note that the simulation environment is just one component of a simulation program or simulator, which sets all the languages to be used in the simulation, similar to a C language developer, defining the functions of printf, for, while, and etc.

The 2D simulation program to be developed in this thesis consists of a server and a client program, which communicate with each other, constantly updating the state of the environment. The server program can be conceived as a soccer league with a specific set of rules and regulations while the client program can be conceived as a team of players with a common gameplay strategy in mind. Simply speaking, the server program provides the simulation environment and the client program provides
the inputs to the simulation environment which are then processed to produce outputs. The server program has a set of pre-defined functions that must be used by the client in order to run the simulation. This is, without question, a simplification in the virtual world. The actual composition and architecture of the program will be explained in greater detail in the next section.
2 Program Theory & Logic

2.1 Program Architecture

Figure III shows the basic program architecture used in the simulation program that has been developed in this thesis. The Controller Unit, Simulation Environment, and Computer Vision System modules, shown in Figure 1, form the server program, and Team A and B Strategy at the top form the client program. The Blackboard belongs neither to the server program nor the client program. The simulation program runs at 10 fps, that is, a refresh rate of 100 ms.

2.2 Module Characteristics

2.2.1 Blackboard

This module functions as a medium for exchange of the most up-to-date information only. It does not directly read anything from other modules nor send anything to other modules. It behaves no different than a physical blackboard, where information on the blackboard is only read and/or written by a specific group of individuals. Note also that the physical blackboard doesn’t read nor write anything itself. The information stays on the blackboard unless erased and rewritten, which is identical to the Blackboard module in the program. There are three sub-modules inside the Blackboard module: Team A Dynamics, Team B Dynamics, and Ball Dynamics. Both Team A and B Dynamics contain three set of data respectively: (1) Speeds, (2) Actions, and (3) Coordinates. Ball Dynamics, however, is determined by the data, or specifically the Actions, in Team A and B Dynamics, and in addition,
contains the ball coordinates. These data will be used in the simulation program, which will be explained in greater detail in the next section of this paper.

2.2.2 Controller Unit

This module reads the Blackboard and moves the robots as specified by the clients. It has three sub-modules: Team A Controller, Team B Controller, and Ball Controller. Team A and B Controller move the two teams of robots and the Ball Controller takes care of the ball. They control the robots by writing to the Simulation Environment.

2.2.3 Simulation Environment

This module generates a graphical output, that is, the simulation environment, to be visualized on the computer screen. The graphical output is generated by Flash. Like the Blackboard, it does not directly read anything from other modules nor send anything to other modules. It runs at 10 fps.

2.2.4 Computer Vision System

This module reads the Simulation Environment, getting the most up-to-date of information from the environment and writing it to the Blackboard. This module completes the loop and provides information for next cycle of analysis for the client-end.
2.2.5 Team A and B Strategy

These two modules are separate and they do not communicate with each other. They communicate with the server only, or specifically the Blackboard. They read and write to the Blackboard in a user-defined time interval. Each of these modules should have at least one strategy to be executed in the simulation. Ideally, they should have more than one strategy to allow for the robots to select the best strategy depending on the state of the environment. Each module analyzes the state of the environment based on its algorithm with a list of speeds and actions as the output.

Note: Table 1 and 2 summarize the read/write capabilities of the modules and the communication connections among modules.
2.3 Program Flow

Figure IV shows the program flow for the simulation program. Some of the modules shown in the program architecture, such as the Blackboard and Controller Unit, are neglected in the figure as they do not exist in reality. They are present in the program architecture for illustration purposes only. However, one can compare Figure 2 and 3 and easily relate each of the blocks in Figure 3 to a specific module. The middle portion of the figure, where there are three blocks (Team A File, Coords File, and Team B File) not enclosed by a dotted line, are actually the Blackboard. And the Team A and B Controller as well as the Ball Controller directly below are the sub-modules in the Controller Unit that is not physically shown.

2.3.1 Program Startup

There are three start buttons in the simulation program. One of them belongs to the server program and two to the client program. For proper operation, one should first initiate the server program using the start button and then initiate the client program since launching the client program is meaningless without the server program in operation. Although the client program, which consists of both Team A and B Strategy, ideally should start at the same time, it is really up to the users’ preference. However, note that as soon as the server program is initiated, the simulation is started, or in other words, the clock is clicking.
2.3.2 Program Operation

When the simulation program is started, Team A and B Strategy will first write to their corresponding team file containing Speeds and Actions. The file contains the desired speed of x and y for each of the robots in the team and the force at which the kicking or passing will be applied. It is important to note that that the robots will not move or take any actions at this stage. Only the instructions are stored in their corresponding team file. Team A and B Strategy will constantly communicate with their corresponding team file during the match. The team file is not accessible by the other team since they reveal the team’s strategy.

At the middle of the figure, there is a block named Coords File, which stores the coordinates of each of the robots as well as the ball. These coordinates are recorded by the Computer Vision System. All these coordinates are accessible by both teams, just as in a real soccer game where a player can see the position of his team’s players as well as the opposing team’s, not to mention the location of the ball. Team A and B Strategy will also constantly communicate with this file containing the coordinates to get feedbacks from the environment for strategy adjustments or modifications during the match.

Team A and B Controller as well as Ball Controller will then read from their corresponding file containing all the instructions. Team A Controller will read from Team A File and Team B Controller will read from Team B File. Ball Controller will read from both Team A and B File, specifically the Actions that are stored in the file. All these instructions are then processed and outputted to the Simulation Environment.
at 10 fps.

Afterwards, the Computer Vision System then comes into play recording the coordinates of each of the robots as well as the ball. This is necessary as there must be some feedbacks provided back to the client program otherwise the users will never know what is happening in the Simulation Environment. However, the Computer Vision System cannot write directly to the file storing the coordinates. This is due to the use of ActionScript in the construction of the program. ActionScript is a very secure programming language, which does not allow writing to any types of files. As a result, a conversion to PHP script with Apache is necessary for any writing to occur. Technically, this conversion is also necessary if Team A and B strategy are written in ActionScript. The conversion is neglected in Figure 3 simply because ActionScript is rarely used when there is no graphical generation, as in Team A and B Strategy. A more widely known language, such C or C++, would most likely be used.

2.3.3 Blackboard – The Significance

A very important component to ensure smooth operation in the simulation program is the Blackboard, which belongs neither to the server nor the client program. The Blackboard not only serves as a medium for exchange of the most up-to-date information, it also helps synchronize the server program with the client program. As mentioned earlier, the server program refreshes every 100 ms, and the client program could refresh faster or slower than 100 ms. If the refresh rate of the client program is faster than 100 ms, there will be an overflow of data causing instability or
execution errors in the simulation program. The Blackboard solves the problem by allowing the server and the client to refresh at a different rate. The client program could refresh and write to the Blackboard every 10 ms or every 200 ms while the server program refreshes itself at its default rate of 100 ms since the difference is eliminated by the Blackboard. Figure V illustrates the significance of the Blackboard.
3 Final Program

3.1 Module Functions

In this section, the specific functions performed in the program by each module described earlier in section 2 are explained. These modules include: graphical simulation, robot controllers, ball controller, camera vision system, and the blackboard.

3.1.1 Graphical Simulation

The graphical simulation is the primary tool to achieve efficient strategic modeling as well as reduce costs during the developmental phase. Graphical simulation enables the user to see and monitor their strategy in action versus a predefined opponent strategy, in which they are trying to counter. Since the field can be transformed into a 2-dimensional topologic view (i.e. bird's eye view from the top), the simplest method to display where the robots and ball are at is by using an x-y coordinate system. The graphical simulation for the SSL consists of two teams of five robots, the ball, and the field. Robots are represented using squares of 20 pixels in width and length. These are scaled to the field by a ratio of “8.71 mm to 1 pixel”. Furthermore, the two teams of robots are differentiated using the colours blue and red. A yellow circle measuring 8 pixels in diameter represents the ball in the simulation. Last but not least, the field is measured 700 pixels in length by 480 pixels in width, and includes an abbreviated set of field markings, which are: kick-off circle, midfield line, and penalty area (see Figure VI).
Graphical simulation is performed on a frame-by-frame basis using Flash. For the purpose of the program, the frames are generated at 10 frames per second (fps). At this speed, computers without superb computational power should still be able to execute the simulation, as well as the client-end algorithms simultaneously. The amount of frames generated per second is set by a timer delay in the program, which reads information from the blackboard at intervals of 100 milliseconds. In order to generate a graphical output on a 2-dimensional surface, two pieces of information is required from the client-end: robot velocity (i.e. commanding robot’s movement), and robot’s kicking power (i.e. commanding ball momentum). This information will be written onto the blackboard, whereby the server will read off and generate the next set of robot coordinates based on the commands; as well, the server generates the next set of ball coordinates based on the intended actions. How these values are generated is done using two controller modules, which are the robot controller module and the ball controller module. As their name states, the robot controller performs the calculations involved in obtaining the set of robot coordinates in the current frame of the graphical output. Similarly, the ball controller performs the calculations involved in obtaining the set of ball coordinates in the current frame of the graphical output. Methodologies behind the two controller modules will be discussed later on. After the coordinates for the robots and balls are generated, these values are set as the coordinates to their respective graphical objects in the program. Ultimately, these values translate to a graphical simulation.
3.1.2 Robot Controllers

Primarily, the robot controllers, one for each team, are used to read in values from the blackboard and turn them into the respective values of robot coordinates and orientation. This is done via two main procedures: obtaining the robot movement data from the blackboard, and generating new values based on the data obtained. Generating new coordinates require the use of simple kinematics and few simple assumptions, whereas robot turning is straightforward. Firstly, assumptions used in modeling the frame-by-frame movement of the robots are:

- friction is neglected due to the short time frame of movement
- no acceleration occurs during the frame of calculation, hence, constant speed
- robots can move in any direction without adjusting the body towards movement direction
- since information sent to the blackboard may not be at 100ms intervals, the robot controller uses the original movement data for calculations instead of stopping the robot when no new data is received

After defining the assumptions, the next step was to develop kinematics equations for movement and turning, which were:

\[
x_{\text{new}} = x_{\text{old}} + \text{“scale factor”}(v_x \times 0.1s) \quad \text{[Eq. 1]}
\]

\[
y_{\text{new}} = y_{\text{old}} + \text{“scale factor”}(v_y \times 0.1s) \quad \text{[Eq. 2]}
\]

movement data is taken in as speed values for the x and y directions (in units of pixels/s). Turning data is taken in as a rotational speed value (in units of deg/s), either positive for counter-clockwise or negative for clockwise, and it would be passed through the equation to obtain the next value for robot orientation using the equation:
\[ \theta_{\text{new}} = \theta_{\text{old}} + (r \times 0.1s) \quad \text{[Eq. 3]} \]

Since the graphical output is generated at the rate of 10 fps, each movement would have a duration of 0.1s, which is the logic behind the term the multiplying term 0.1s in the equations 1, 2, and 3. Once the dataset for the new frame has been generated, the robot controller would assign the values to the corresponding graphical robot objects.

Another feature embedded into the robot controller module is a collision test between the robots. Due to the complexity of determining which side is at fault when a collision occurs, the program would simply halt when this occurs. In the actual SSL, a free kick would be awarded to the team not at fault, but since the main objective of this program is to test a specific strategy, there should be no need for refereeing. However, due to the autonomous nature of the robots, minor collisions might occur, and if that is the case with certain strategies, this feature can be toggled off. Methodology behind this function is quite simple from a programming standpoint. First of all, the condition for a collision is when a robot’s physical entity comes into contact with another robot. In mathematics with reference to the x-y coordinate system, a collision occurs when any one coordinate of the bounded set of points corresponding to the physical entity of the robot (i.e. [x ± a, y ± a], whereby ‘a’ is the side length of the robot object divided by 2, and (x, y) refers to the centre of the robot) overlaps with any one coordinate of the bounded set of points of an opponent robot. In order to test this, every time the robot controller calculates a new set of robot coordinates, it runs the values through the robot collision test.
More specifically, it runs through the collision test in a loop for each robot, and checks if the centre-to-centre distance between any two robots on opposing sides would be less than 20 pixels (i.e. the minimum length two robots could be apart without colliding). Furthermore, the equation used for distance verification is:

\[
\text{distance} = \left[ (x_{\text{robot1}} - x_{\text{robot2}})^2 + (y_{\text{robot1}} - y_{\text{robot2}})^2 \right]^{1/2}
\]  

[Eq. 4]

When a collision condition is satisfied, the program returns a value indicating the colliding robots, and halts the simulation. If no collision occurs, the robot controller assigns the x-y coordinate values to the corresponding robot objects.

3.1.3 Ball Controller

The ball controller’s main task is to generate realistic ball movement in the simulation. Similar to the robot controller, the ball controller generates the x-y coordinates for the ball from the ball movement data obtained from the blackboard. In order to incorporate realistic ball movement into the simulation, dynamics modeling was used. For dynamics modeling, the client-end input for ball movement would be a momentum value in the x and y directions. Assumptions made for the dynamics model were:

- friction factor is constant
- momentum decreases with the friction factor at each frame
- there is absolutely no spin on the ball

Having determined the assumptions, the equations used to obtain the next set of x-y coordinate pair for the ball would be:
\begin{align*}
\text{Eq. 5a} & \quad x_{\text{new}} = x_{\text{old}} + (m \times f) \\
\text{Eq. 5b} & \quad y_{\text{new}} = y_{\text{old}} + (m \times f)
\end{align*}

where ‘m’ is the momentum value imparted to the ball, and f is the friction factor.

After obtaining the new x and y ball coordinates, the program puts the coordinate pair through a collision test. But before even momentum is imparted to the ball, the program has a condition verifying if the kicking robot is able to actually kick the ball. Mathematically, the centre-to-centre distance between the kicking robot to the ball must be within a range where kicking is possible. If this was not the case, the program would halt and warn the user of an error with wrongful kicking. Another condition is to check if the ball is outside the bounds of the field. When both conditions pass, the ball controller would enter the second phase of the module, which is the collision test.

The collision test evaluates if the ball has collided with another physical entity (i.e. a robot). Similar to the robot controller, the collision test runs through a loop to test if the ball collides with any of the robots. If this occurs, realistically, the ball would bounce off the object with a loss of momentum. In the program, to simplify this complicated model, few assumptions were made, which were:

- momentum loss is neglected from the collision
- there is no spin on the ball
- ball reflects off the object’s side in a way such that the angle of incidence (incoming angle) is equal to the angle of reflection (outgoing angle), please see Figure VI for graphical explanation

Modeling this action mathematically requires more effort, requiring the use of
trigonometry. This entire process takes part in two functions, which are determining if a collision has occurred, and what angle should the ball reflect back at. How the ball should reflect back requires a much more complicated logic, which will be discussed separately from the ball controller module. After accounting for collision and reflection, the obtained values are assigned as the new x-y coordinate pairs for the ball in the current frame.

3.1.4 Ball Rebound Model

In order to assess how the ball would rebound (assuming it would follow the way as described in Figure VI), the slope of the ball as well as slopes of the sides of the colliding robot must be known. Since the robot’s center x, y coordinate values are known and the sides are of constant length of 20 pixels, the following formulas are used to identify the 4 corner coordinates of the robot (see Figure VII for graphical explanation):

- \( x_{c1} = x_{\text{robot}} + [k \times \cos(45^\circ - \theta_{\text{robot}})] \)  \[\text{Eq. 6a}\]
- \( y_{c1} = y_{\text{robot}} - [k \times \sin(45^\circ - \theta_{\text{robot}})] \)  \[\text{Eq. 6b}\]
- \( x_{c2} = x_{c1} - [k \times \sin(\theta_{\text{robot}})] \)  \[\text{Eq. 7a}\]
- \( y_{c2} = y_{c1} + [k \times \cos(\theta_{\text{robot}})] \)  \[\text{Eq. 7b}\]
- \( x_{c3} = x_{c2} - [k \times \sin(\theta_{\text{robot}})] \)  \[\text{Eq. 8a}\]
- \( y_{c3} = y_{c2} - [k \times \cos(\theta_{\text{robot}})] \)  \[\text{Eq. 8b}\]
- \( x_{c4} = x_{c3} + [k \times \sin(\theta_{\text{robot}})] \)  \[\text{Eq. 9a}\]
- \( y_{c4} = y_{c3} - [k \times \cos(\theta_{\text{robot}})] \)  \[\text{Eq. 9b}\]
where $c_1$ is corner 1, $c_2$ is corner 2, etc. and $k$ is 14.14214. After having obtained
the corner values, the program also keeps track of two sets of ball x-y coordinate
values; the set for the current frame and the set for the previous frame. Since the
collision point (hereby known as “intersection point”) requires two lines to cross,
the two lines being verified in the ball rebound model are: the ball’s trajectory and the
line representing the colliding side of the robot. A slope value is generated from
each set of points, and in total, there are three slope values used in the process: ball
trajectory’s slope, one slope for each parallel side pairs of the robot. Logic behind
solving the intersection point is simply solving the system of equations:

- $\text{slope}_{\text{ball}} x (x_{\text{int}} - x_{\text{ball}}) = y_{\text{int}} - y_{\text{ball}}$ [Eq. 10]
- $\text{slope}_{\text{side}} x (x_{\text{int}} - x_c) = y_{\text{int}} - y_c$ [Eq. 11]

where equations 10 and 11 are the lines for the ball trajectory and for the robot side,
respectively. Once the intersection point is found, the next step is to determine the
new angle at which the ball exits after collision. This is done by first solving for the
slope of the line perpendicular to the collision side (also known as the normal) at the
point of intersection. Slope of the normal is simply:

- $\text{slope}_{\text{normal}} = -1/\text{slope}_{\text{side}}$ [Eq. 12]

After knowing the equation of the normal, one can find the equation of the line
perpendicular to the normal (knowing the slope is the same as the colliding side’s
slope) intersecting with any arbitrary point, $P$, on the ball’s trajectory (see Figure IX).
The perpendicular distance between $P$ to the normal would be the same as the
mirrored point, $P'$ on the reflected trajectory to the normal. Hence, by locating the
point at the same distance away from the normal, and using it in conjunction with the intersection point, the slope of the reflected line is obtained. Since the slope of any line is the tangent value of the angle it forms with the $x$-axis in an $x$-$y$ coordinate system, the angle is obtained by:

$$\text{angle}_{\text{reflection}} = \arctan(\text{slope}_{\text{reflection}}) \quad [\text{Eq. 13}]$$

With the conservation of momentum in mind (assuming the ball momentum magnitude remains the same), the new $x$ and $y$ values of ball momentum after the reflection would be:

- $\text{ballmomentum}_x = \text{ballmomentum}_{\text{original}} \times \cos(\text{angle}_{\text{reflection}}) \quad [\text{Eq. 14a}]$
- $\text{ballmomentum}_y = \text{ballmomentum}_{\text{original}} \times \sin(\text{angle}_{\text{reflection}}) \quad [\text{Eq. 14b}]$

Due to the complexity in the math, the actual code for calculating the ball momentum after reflection was not completed, however, the identification for which side the ball collides at is included.

### 3.1.5 Camera Vision System

Since this simulation was assumed to be employed by teams using a central-commanding computer in conjunction with an overhead camera vision system, it was logical to also introduce a similar simulated function into the program. As described earlier, the camera vision system allows the teams to record what is taking place on the field from an overhead position. Typically, the camera used can take pictures at 25 – 75fps, and send them wirelessly to the central computer [4]. However, the time it takes to process the image and identify the objects and their
absolute positions could take well over 40ms, which is at the rate of about 25fps [5]. The picture taken by the camera would be sent back to the central computer, which identifies the robots and their locations. From these values, the team’s algorithm would generate the next set of robot actions.

In order to simulate this function in the program, the camera vision function takes in the coordinate values for all the objects on the field, and writes all the values to the blackboard, where the client-end algorithm could read the new set of data. To account for what transpires in real-game settings, the user is capable of adjusting the rate at which the camera’s data is fed onto the blackboard. Reason for this is to account for the lag in image processing time inherent in any camera vision system. Since this value varies for various image processing algorithms and setups, the user is able to modify this value to approach a more realistic environment during actual competition. Furthermore, the simulation runs at 10fps, which means the camera vision system should report the coordinates at frequencies less than 10fps (i.e. actual gameplay speed is real-time, where camera vision is pseudo real-time and has to be less ‘frequent’ than gameplay speed). Keeping this in mind, the rate at which the camera returns values in the program can be varied at 100ms intervals, which is the same as the time per frame generated in the program. Ultimately, this allows the user to vary the time at which the camera returns field data to approach a more realistic environment such as the one during the competition.
3.1.6 Blackboard

This is the module where values for coordinates, robot movement speed, robot kick power, and etc. are located. As explained earlier, the blackboard serves as a medium of information exchange between the client-end and server-end. Conceptually, this is similar to a blackboard where the teacher writes to and the student reads from, but neither could communicate directly to each other. Putting this concept into the program requires the use of three text files. Each side uses one text file to store the robot action data, and the third file is used by the camera vision system to write the coordinate data. Each file would contain a string of variables relating to the various properties of each object such as robot movement velocities, robot kicking power, and robot angular velocity. The specific names of each variable recognized by the program are listed in Tables 3, 4, and 5. These text files are repeated erased and written on each time the client-end writes in the desired actions or when the server-end writes in the new set of vision data. While the simulation is running, the text files will always contain the latest set of written data, this is similar to a real-game scenario where robots would perform its actions from its latest command. It is important to note that to write the camera vision outputs to the blackboard requires the use of PHP script. The files used were modified versions of ones used in an online Flash tutorial site [6].
3.2 Program Interface

3.2.1 User Customizability

Due to the complexity to simulate the actual competition environment, the program features a customizable set of variables for the user (see Figure X). The three customizable variables are: simulation frame rate, vision system frame rate, and ball friction factor. Simulation frame rate is defined as the time between each output frame generated in the graphical simulation. By default, the program sets the output frame rate at 100ms. Unless the computer running both the client-end algorithm and the server has sufficient computing power, the simulation frame rate should not go any lower than 100ms, but can be adjusted upwards. The vision system frame rate is the speed at which the new set of coordinates of all the objects on the field are converted to x-y values and written onto the blackboard. This is to approximate in actual competition the lag occurring between the camera vision system and the real-time situation. In the program, this value is adjusted as a multiple of the simulation frame rate. For example, the user could adjust the vision system’s rate at four times the simulation rate, which means that for every four frames generated as graphical output, one dataset would be written onto the blackboard via the vision system function. Last but not least, the user has the ability to change the friction factor, which from the program’s perspective, only affects the ball’s movement. Values for the friction factor ranges from 0 to 1 (all real values in the range is valid), where 0 is the case of extreme friction (i.e. ball would not move at all), and 1 is the case of no friction. Users are encouraged to test out various friction
factors to adjust for actual field conditions.
4 Conclusion

4.1 Final Product

Using Flash, a simple and efficient way for strategy testing for the RoboCup SSL was created. Users could run through their AI algorithm through the simulator, and see how their strategies would behave in various conditions. Furthermore, it does not require top of the line computing power due to the relatively simplistic codes used throughout. Even though the codes are simple, customizability as well as various actual game-like physics were incorporated into the simulation. Various parameters are alterable in the program, allowing the user to approach realistic perform through the simulation. Nonetheless, there are various improvements requiring still much work, but those who appreciate this program and would like to improve on the existing program could easily do we have open-sourced the code. In conclusion, the program not only serves as an efficient medium for strategy testing, but also a framework for further research in creating the perfect simulator for any RoboCup event.

4.2 Future Improvements

Looking towards the future, some areas of improvement for the program would be: a better interface to link client-end algorithm to the server, the physics modeling used for ball movement, introduce noise and fuzziness in the robots, and improved robot models. Firstly, a better interface linking the client-end algorithm to the server facilitates the loading of the two teams’ algorithms. Second, the current physics
modeling for ball movement could be improved with more advanced functions such as ball spin, and ball airtime. For example, from watching past videos, SSL robots are able to cross the ball from corner kicks. However, the program does not incorporate any airtime physics into the ball movement aspect. Third, in reality, noise and fuzziness exist in any mechatronics system, hence, they never accurately perform what they are signaled to. This noise generates various random issues, but is also one challenge which the team must overcome. By incorporating this into the program, the simulator is one step closer to approaching realistic competition conditions. Lastly, current robot models are simple squares representing only the physical boundaries. However, in reality, these robots usually have a kicker mechanism at one of its sides (usually at the front). If the kicker mechanism is considered when the robot decides to kick, the robot would have to adjust its kicker inline with the ball before imparting a momentum to it. This creates realism as well as another level of challenge for the client’s algorithm. Ultimately, these improvements would enable the simulator to approach competition-like realism via software modeling.
Collection of Figures

Figure I Sample screenshot of simulation

Figure II SSL Field Dimensions (in millimetres)
Figure III Program architecture
B cannot process information fast enough than A can generate.

A can generate data as fast as it can to an intermediate X, and B can process at a slower rate by taking the data from the intermediate X every 100 ms.

Figure V Blackboard significance
Figure VI Simulation Field Dimensions (in pixels)

Figure VII Ball reflection assumption

θ_{incident} \quad θ_{reflected}

Surface (maybe be offset at various angles, currently it has no offset)
Figure VIII Explanation for obtaining robot corner coordinates

- Parallel to collision side (known slope)
- Point on reflected trajectory, at an equal distance away from the normal as the arbitrary point (unknown x,y)
- Intersection point (known x,y)
- Arbitrary point on ball trajectory (known x,y)
- Normal (slope)

Figure IX Explanation for obtaining reflection angle
Figure X Simulation interface with user customizable variations

Figure XI Sample frames in simulation
### Table 1 Module Read/Write Capabilities

<table>
<thead>
<tr>
<th>Modules</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team A Strategy</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Team B Strategy</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Blackboard</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Controller Unit</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Simulation Environment</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Computer Vision System</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Table 2 Module-to-Module Communication

<table>
<thead>
<tr>
<th>Variable name in blackboard (case-sensitive)</th>
<th>Variable definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr1x</td>
<td>x-direction speed of blue team robot 1</td>
</tr>
<tr>
<td>fr1y</td>
<td>y-direction speed of blue team robot 1</td>
</tr>
<tr>
<td>fr1r</td>
<td>angular speed of blue team robot 1</td>
</tr>
<tr>
<td>fr1bmx</td>
<td>x-direction ball momentum imparted by blue team robot 1</td>
</tr>
<tr>
<td>fr1bmy</td>
<td>y-direction ball momentum imparted by blue team robot 1</td>
</tr>
<tr>
<td>fr2x</td>
<td>x-direction speed of blue team robot 2</td>
</tr>
<tr>
<td>fr2y</td>
<td>y-direction speed of blue team robot 2</td>
</tr>
<tr>
<td>fr2r</td>
<td>angular speed of blue team robot 2</td>
</tr>
<tr>
<td>fr2bmx</td>
<td>x-direction ball momentum imparted by blue team robot 2</td>
</tr>
<tr>
<td>fr2bmy</td>
<td>y-direction ball momentum imparted by blue team robot 2</td>
</tr>
<tr>
<td>fr3x</td>
<td>x-direction speed of blue team robot 3</td>
</tr>
<tr>
<td>fr3y</td>
<td>y-direction speed of blue team robot 3</td>
</tr>
<tr>
<td>fr3r</td>
<td>angular speed of blue team robot 3</td>
</tr>
<tr>
<td>fr3bmx</td>
<td>x-direction ball momentum imparted by blue team robot 3</td>
</tr>
<tr>
<td>fr3bmy</td>
<td>y-direction ball momentum imparted by blue team robot 3</td>
</tr>
<tr>
<td>fr4x</td>
<td>x-direction speed of blue team robot 4</td>
</tr>
<tr>
<td>fr4y</td>
<td>y-direction speed of blue team robot 4</td>
</tr>
<tr>
<td>fr4r</td>
<td>angular speed of blue team robot 4</td>
</tr>
<tr>
<td>fr4bmx</td>
<td>x-direction ball momentum imparted by blue team robot 4</td>
</tr>
<tr>
<td>fr4bmy</td>
<td>y-direction ball momentum imparted by blue team robot 4</td>
</tr>
</tbody>
</table>

### Table 3 Variable names for robot commands (opponent side replaces the ‘f’ with ‘o’)

<table>
<thead>
<tr>
<th>Variable name in blackboard (case-sensitive)</th>
<th>Variable definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>fo1x</td>
<td>x-direction speed of blue team robot 1</td>
</tr>
<tr>
<td>fo1y</td>
<td>y-direction speed of blue team robot 1</td>
</tr>
<tr>
<td>fo1r</td>
<td>angular speed of blue team robot 1</td>
</tr>
<tr>
<td>fo1bmx</td>
<td>x-direction ball momentum imparted by blue team robot 1</td>
</tr>
<tr>
<td>fo1bmy</td>
<td>y-direction ball momentum imparted by blue team robot 1</td>
</tr>
<tr>
<td>fo2x</td>
<td>x-direction speed of blue team robot 2</td>
</tr>
<tr>
<td>fo2y</td>
<td>y-direction speed of blue team robot 2</td>
</tr>
<tr>
<td>fo2r</td>
<td>angular speed of blue team robot 2</td>
</tr>
<tr>
<td>fo2bmx</td>
<td>x-direction ball momentum imparted by blue team robot 2</td>
</tr>
<tr>
<td>fo2bmy</td>
<td>y-direction ball momentum imparted by blue team robot 2</td>
</tr>
<tr>
<td>fo3x</td>
<td>x-direction speed of blue team robot 3</td>
</tr>
<tr>
<td>fo3y</td>
<td>y-direction speed of blue team robot 3</td>
</tr>
<tr>
<td>fo3r</td>
<td>angular speed of blue team robot 3</td>
</tr>
<tr>
<td>fo3bmx</td>
<td>x-direction ball momentum imparted by blue team robot 3</td>
</tr>
<tr>
<td>fo3bmy</td>
<td>y-direction ball momentum imparted by blue team robot 3</td>
</tr>
<tr>
<td>Variable name in blackboard (case-sensitive)</td>
<td>Variable definition</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>frgtx</td>
<td>x-direction speed of blue team goaltender robot</td>
</tr>
<tr>
<td>frgty</td>
<td>y-direction speed of blue team goaltender robot</td>
</tr>
<tr>
<td>frgr</td>
<td>angular speed of blue team goaltender robot</td>
</tr>
<tr>
<td>frgtbgmx</td>
<td>x-direction ball momentum imparted by blue team goalie</td>
</tr>
<tr>
<td>frgtbgmy</td>
<td>y-direction ball momentum imparted by blue team goalie</td>
</tr>
</tbody>
</table>

Table 4 Variable names for robot commands continued (opponent side replaces the ‘f’ with ‘o’)

<table>
<thead>
<tr>
<th>Variable name in blackboard (case-sensitive)</th>
<th>Variable definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1x</td>
<td>x-coordinate value of blue team robot 1</td>
</tr>
<tr>
<td>f1y</td>
<td>y-coordinate value of blue team robot 1</td>
</tr>
<tr>
<td>f2x</td>
<td>x-coordinate value of blue team robot 2</td>
</tr>
<tr>
<td>f2y</td>
<td>y-coordinate value of blue team robot 2</td>
</tr>
<tr>
<td>f3x</td>
<td>x-coordinate value of blue team robot 3</td>
</tr>
<tr>
<td>f3y</td>
<td>y-coordinate value of blue team robot 3</td>
</tr>
<tr>
<td>f4x</td>
<td>x-coordinate value of blue team robot 4</td>
</tr>
<tr>
<td>f4y</td>
<td>y-coordinate value of blue team robot 4</td>
</tr>
<tr>
<td>fgtx</td>
<td>x-coordinate value of blue team robot goalie</td>
</tr>
<tr>
<td>fgty</td>
<td>y-coordinate value of blue team robot goalie</td>
</tr>
<tr>
<td>bx</td>
<td>x-coordinate value of ball</td>
</tr>
<tr>
<td>by</td>
<td>y-coordinate value of ball</td>
</tr>
</tbody>
</table>

Table 5 Variable names for camera vision system coordinates (opponent side replaces the ‘f’ with ‘o’)

36
References


Appendix A: Main Program Source Code

```javascript
//boolean to verify if robotCtrl has done its routine
_global.greenlight = true;

//number of frames per camera shot
_global.inputframerate = 5;

//simulation frame rate
_global.intervaltimer = 100;

//counter for number of frames between camera shots
_global.inputframecounter = 0;

//blue team robots variables to demonstrate simulation
_global.frobot1x = 1;
_global.frobot1y = 1;
_global.frobot2x = 1;
_global.frobot2y = 1;
_global.frobot3x = 1;
_global.frobot3y = 1;
_global.frobot4x = 1;
_global.frobot4y = 1;

//red team robots variables to demonstrate simulation
_global.orobot1x = 0;
_global.orobot1y = 0;
_global.orobot2x = 0;
_global.orobot2y = 0;
_global.orobot3x = 0;
_global.orobot3y = 0;
_global.orobot4x = 0;
_global.orobot4y = 0;

//initialize ball variables
_global.ballx = ball._x;
_global.bally = ball._y;
_global.balloldx;
_global.balloldy;
//keeps the previous ballmomentumx value
_global.oldballmomentumx;
//keeps the previous ballmomentumy value
_global.oldballmomentumy;
//ball friction factor default setting
_global.ballfriction = 0.1;

var intervalt:Number;
var frobotarray:Array = new Array(frobot_1, frobot_2, frobot_3, frobot_4, frobot_gt);
var orobotarray:Array = new Array(orobot_1, orobot_2, orobot_3, orobot_4, orobot_gt);

//interval is a dummy movieclip object holding the coordinates for friendly team
this.createEmptyMovieClip("frobot", this.getNextHighestDepth());

interval.onChanged = function(){
    visionmultiplier.text = "x " + intervalt.text + "ms";
}

stopbutton.onRelease = function(){
    inputframerate = visionfreq.text; //default at 5 simulated frames per camera shot
    ballfriction = friction.text; //default at 0.1
    intervatimer = intervalt.text; //default at 100ms
    writeData();
}

/*readData reads off the blackboard for robot information*/
function readData(){
    n = 0;
    //load variables into frobot array
    //ensures input parameters to the simulation are loaded
    do{
        loadVariables("blueteam.txt", frobot);
    }while(frobot.var6 > 0);
    do{
```
greenlight = false;
robotCtrl();
ballCtrl();
inputframecounter++;
if (inputframecounter == inputframerate){
    visionsystem();
    inputframecounter = 0;
}
}while(greenlight == false);
writeData();

//writeData sets the interval for writing information to the input side of the blackboard
function writeData(){
    if(greenlight == true){
        //set time interval for writeParams to delay its writing
        interval = setInterval(this, "writeParams", intervaltimer);
        //writeParams();
    }else{
        trace("no greenlight");
    }
}

/*writeParams is called by writeData to write information onto the blackboard (this would be the algorithm in the real thing)*/
function writeParams(){
    //string containing the robot movement data, to be written to blackboard
    var frobotmvmt:String = "";
    var i:Number;
    //this following for loop generates the movement variables for the robots
    for(i = 1;i <= 9;i += 1){
        submittedData = new LoadVars();
        submittedData.counter = i;
        switch(i){
        case 1:
            frobotmvmt = frobotmvmt + "fr1x" + "=" + frobot1x + "&";
            break;
        case 2:
            frobotmvmt = frobotmvmt + "fr1y" + "=" + frobot1y + "&";
            break;
        case 3:
            frobotmvmt = frobotmvmt + "fr1bm" + "=" + 6 + "&";
            break;
        case 4:
            frobotmvmt = frobotmvmt + "fr2x" + "=" + frobot2x + "&";
            break;
        case 5:
            frobotmvmt = frobotmvmt + "fr2y" + "=" + frobot2y + "&";
            break;
        case 6:
            frobotmvmt = frobotmvmt + "fr2bm" + "=" + 6 + "&";
            break;
        case 7:
            frobotmvmt = frobotmvmt + "fr3x" + "=" + frobot3x + "&";
            break;
        case 8:
            frobotmvmt = frobotmvmt + "fr3y" + "=" + frobot3y + "&";
            break;
        case 9:
            break;
        }
    }
}
frobotmvmt = frobotmvmt + "fr3bm" + "=" + 6 + ";";
break;

/*case 10 :*/
frobot*

default :
trace("something went wrong...");
break;
}

trace(frobotmvmt);
submittedData.inputData = frobotmvmt;
do{
    submittedData.onload = function(){
        trace(submittedData.verify);
        //triggers readData function after 'resp' has been loaded by PHP script
        readData();
    }
} while (submittedData.verify != success);
//trace(resp.failMsg);
clearInterval(interval);

//robotCtrl places the robots in their respective positions, taken from the blackboard
function robotCtrl(){
    var tempx:Number, tempy:Number;
    frobotarray[0]._x += parseFloat(frobot.fr1x);
    frobotarray[0]._y += parseFloat(frobot.fr1y);
    frobotarray[0]._rotation += 0;
    frobotarray[1]._x += parseFloat(frobot.fr2x);
    frobotarray[1]._y += parseFloat(frobot.fr2y);
    frobotarray[1]._rotation += 0;
    frobotarray[2]._x += parseFloat(frobot.fr3x);
    frobotarray[2]._y += parseFloat(frobot.fr3y);
    frobotarray[2]._rotation += 0;

    if(checkrbtcollision() > 0 and checkrbtcollision() < 200){
        trace("Robots on opposing teams collided");
    }else if(checkrbtcollision() > 200 and checkrbtcollision() < 300){
        trace("Robots on blue team collided");
    }else if(checkrbtcollision() > 300){
        trace("Robots on blue team collided");
    }
greenlight = true;
}

//checks collision among all robots
function checkrbtcollision(){
    var dist:Number;
    var returnvalue:Number;

    //this portion checks between blue team and red team
    //for(i=0;i<=4;i++){+}
    //for(j=0;1<=4;j++){
    dist = Math.sqrt((Math.pow((frobotarray[i]._x - orobotarray[j]._x), 2) + Math.pow((frobotarray[i]._y - orobotarray[j]._y), 2)));
    if(dist < 30){
        returnvalue = i*10 + j + 100;
        return returnvalue;
    }
}

    //this portion checks between the robots on the blue team...
for(i=0;i<=4;i++)
    for(j=0;j<=4;j++)
        if(i<>j)
            dist = Math.sqrt(Math.pow((frobotarray[i]._x - frobotarray[j]._x), 2) + Math.pow((frobotarray[i]._y - frobotarray[j]._y), 2));
        if(dist < 30)
            returnvalue = i*10 + j + 200;
            return returnvalue;
    }
}

//this portion checks between the robots on the red team
for(i=0;i<=4;i++)
    for(j=0;j<=4;j++)
        if(i<>j)
            dist = Math.sqrt(Math.pow((orobotarray[i]._x - orobotarray[j]._x), 2) + Math.pow((orobotarray[i]._y - orobotarray[j]._y), 2));
        if(dist < 30)
            returnvalue = i*10 + j + 300;
            return returnvalue;
    }
}

return -1; //when no collision occurs, function returns -1
}

//this function controls the ball movement
function ballCtrl(){
    var whohasball:Number;
    oldballmomentum = ballmomentum;
    trace(ballmomentum);
    //this portion ensures the kicking robot is near the ball
    whohasball = checkhasball();
    trace("whohasball = " + whohasball);
    if (whohasball == 0){
        ballmomentum = frobot.fr1bm;
    }else if (whohasball == 1){
        ballmomentum = frobot.fr2bm;
    }else if (whohasball == 2){
        ballmomentum = frobot.fr3bm;
    }else if (whohasball == 3){
        ballmomentum = frobot.fr4bm;
    }else if (whohasball == 4){
        ballmomentum = frobot.frgtbm;
    }else if (whohasball == 10){
        ballmomentum = orobot.or1bm;
    }else if (whohasball == 11){
        ballmomentum = orobot.or2bm;
    }else if (whohasball == 12){
        ballmomentum = orobot.or3bm;
    }else if (whohasball == 13){
        ballmomentum = orobot.or4bm;
    }else if (whohasball == 14){
        ballmomentum = orobot.orgtbm;
    }
}

if (isNaN(ballmomentum)) ballmomentum = oldballmomentum; //ensures ballmomentum is still the same even if it hits robot
ballox = ballx;
bally = bally;
bally = ballx + ballmomentum;
bally = bally + ballmomentum;
ballmomentum = ballmomentum * Math.exp(-ballfriction);
if(whohasball >= 0) ballrebound(whohasball);

ball._x = ballx;
ball._y = bally;

if(ballx > 640 or ballx < 0 or bally < 0 or bally > 480){
  trace("ball out of bounds");
}

if(checkballcollision() > 9){
  trace("Ball collided with red team");
}
else if(checkballcollision() > -1){
  trace("Ball collided with blue team");
}

//ballrebound simulates the bounce of the ball when it comes into contact of an obstacle
function ballrebound(robotnumber){
  //c1x, c1y so on are the corner coordinates of the robot object
  //k is the diagonal distance from centre of robot to its corner
  //w is the side length of the robot
  var c1x:Number, c1y:Number, c2x:Number, c2y:Number, c4x:Number, c4y:Number, k:Number = 14.14214, w:Number = 20;
  var c3x:Number, c3y:Number, c6y:Number, keyrobot:Object;
  var slope1:Number, slope2:Number, ballslope:Number;
  var dist:Number, shortestdist:Number, closestcorner:Number;

  if(robotnumber >= 10){
    keyrobot = orobotarray[robotnumber - 10]; //the targeted robot is on the red side
  }else if(robotnumber >= 0){
    keyrobot = frobotarray[robotnumber]; //the targeted robot is on the blue side
  }

  c1x = keyrobot._x + k*Math.cos((45-keyrobot._rotation)/57.3);
  c1y = keyrobot._y - k*Math.sin ((45-keyrobot._rotation)/57.3);
  c2x = c1x - w*Math.sin((keyrobot._rotation)/57.3);
  c2y = c1y + w*Math.cos((keyrobot._rotation)/57.3);
  c3x = c2x - w*Math.cos((keyrobot._rotation)/57.3);
  c3y = c2y - w*Math.sin((keyrobot._rotation)/57.3);
  c4x = c3x + w*Math.sin((keyrobot._rotation)/57.3);
  c4y = c3y - w*Math.cos((keyrobot._rotation)/57.3);

  //determine which side the ball hits the robot
  var xint:Number, yint:Number;
  var leftside:Number, rightside:Number;
  var dist = Math.sqrt(Math.pow((ballx-c1x), 2) + Math.pow((bally-c1y), 2));
  shortestdist = dist;
  closestcorner = 1;
  dist = Math.sqrt(Math.pow((ballx-c2x), 2) + Math.pow((bally-c2y), 2));
  if(dist < shortestdist){
    shortestdist = dist;
    closestcorner = 2;
  }
  dist = Math.sqrt(Math.pow((ballx-c3x), 2) + Math.pow((bally-c3y), 2));
  if(dist < shortestdist){
    shortestdist = dist;
    closestcorner = 3;
  }
  dist = Math.sqrt(Math.pow((ballx-c4x), 2) + Math.pow((bally-c4y), 2));
  if(dist < shortestdist){


shortestdist = dist;
closestcorner = 4;
}
if(c1x <= c2x and (closestcorner == 1 or closestcorner == 2)){
xint = c1x;
if(slope1 == Number.POSITIVE_INFINITY or slope1 == Number.NEGATIVE_INFINITY){
    if(c1y < c2y){
        yint = c1y;
        while(yint <= c2y){
            leftside = ballslope * (xint - ballx);
            rightside = yint - bally;
            yint += 0.01;
            if(Math.abs(leftside - rightside) < 0.1){
                //ball rebound physics go here
                return -1;
            }
        }
        xint = c2x + 1; //ensures the program doesn't go to the next redundant part
    }
    else if(c2y < c1y){
        yint = c2y;
        while(yint <= c1y){
            leftside = ballslope * (xint - ballx);
            rightside = yint - bally;
            yint += 0.01;
            if(Math.abs(leftside - rightside) < 0.1){
                //ball rebound physics go here
                return -1;
            }
        }
        xint = c2x + 1; //ensures the program doesn't go to the next redundant part
    }
}
while(xint <= c2x){
    xint += 0.01;
    yint = slope1 * xint - (slope1 * c2x) + c2y;
    leftside = ballslope * (xint - ballx);
    rightside = yint - bally;
    if(Math.abs(leftside - rightside) < 0.1){
        //ball rebound physics go here
        return -1;
    }
}
}
}
else if(c2x < c1x and (closestcorner == 1 or closestcorner == 2)){
xint = c2x;
while(xint <= c1x){
    xint += 0.01;
    yint = slope1 * xint - (slope1 * c1x) + c1y;
    leftside = ballslope * (xint - ballx);
    rightside = yint - bally;
    if(Math.abs(leftside - rightside) < 0.1){
        //ball rebound physics go here
        return -1;
    }
}
}
if(c2x <= c3x and (closestcorner == 2 or closestcorner == 3)){
xint = c2x;
if(slope2 == Number.POSITIVE_INFINITY or slope2 == Number.NEGATIVE_INFINITY){
    if(c2y < c3y){
        yint = c2y;
        while(yint <= c3y){
            leftside = ballslope * (xint - ballx);
            rightside = yint - bally;
            yint += 0.01;
            if(Math.abs(leftside - rightside) < 0.1){
                //ball rebound physics go here
                return -1;
            }
        }
        xint = c3x + 1; //ensures the program doesn't go to the next redundant part
    }
    else if(c3y < c2y){
        yint = c3y;
        while(yint <= c2y){
            leftside = ballslope * (xint - ballx);
            rightside = yint - bally;
            yint += 0.01;
            if(Math.abs(leftside - rightside) < 0.1){
                //ball rebound physics go here
                return -1;
            }
        }
        xint = c3x + 1; //ensures the program doesn't go to the next redundant part
    }
}

leftside = ballslope * (xint - ballx);
rightside = yint - bally;
yint += 0.01;
if(Math.abs(leftside - rightside) < 0.1) {
    //ball rebound physics go here
    return -1;
}
xint = c3x + 1;  //ensures the program doesn't go to the next redundant part
}
else if(c3y < c2y) {
    yint = c3y;
    while(yint <= c2y) {
        leftside = ballslope * (xint - ballx);
        rightside = yint - bally;
        yint += 0.01;
        if(Math.abs(leftside - rightside) < 0.1) {
            //ball rebound physics go here
            return -1;
        }
    }
    xint = c3x + 1;  //ensures the program doesn't go to the next redundant part
}
}
while(xint <= c3x) {
    xint += 0.01;
    yint = slope2 * xint - (slope1 * c3x) + c3y;
    leftside = ballslope * (xint - ballx);
    rightside = yint - bally;
    if(Math.abs(leftside - rightside) < 0.1) {
        //ball rebound physics go here
        return -1;
    }
}
}
else if (c3x < c2x and (closestcorner == 2 or closestcorner == 3)) {
    xint = c3x;
    while(xint <= c3x) {
        xint += 0.01;
        yint = slope2 * xint - (slope1 * c3x) + c2y;
        leftside = ballslope * (xint - ballx);
        rightside = yint - bally;
        if(Math.abs(leftside - rightside) < 0.1) {
            //ball rebound physics go here
            return -1;
        }
    }
    xint = c3x + 1;  //ensures the program doesn't go to the next redundant part
}
}

if(c3x <= c4x and (closestcorner == 3 or closestcorner == 4)) {
    xint = c3x;
    if(slope1 == Number.POSITIVE_INFINITY or slope1 == Number.NEGATIVE_INFINITY) {
        if(c3y < c4y) {
            yint = c3y;
            while(yint <= c4y) {
                leftside = ballslope * (xint - ballx);
                rightside = yint - bally;
                yint += 0.01;
                if(Math.abs(leftside - rightside) < 0.1) {
                    //ball rebound physics go here
                    return -1;
                }
            }
            xint = c4x + 1;  //ensures the program doesn't go to the next redundant part
        }
    }
}
else if(c4y < c3y){
    yint = c4y;
    while(yint <= c3y){
        leftside = ballslope * (xint - ballx);
        rightside = yint - bally;
        yint += 0.01;
        if(Math.abs(leftside - rightside) < 0.1){
            //ball rebound physics go here
            return -1;
        }
    }
    xint = c4x + 1;  //ensures the program doesn't go to the next redundant part
}
}

while(xint <= c4x){
    xint += 0.01;
    yint = slope1 * xint - (slope1 * c4x) + c4y;
    leftside = ballslope * (xint - ballx);
    rightside = yint - bally;
    if(Math.abs(leftside - rightside) < 0.1){
        //ball rebound physics go here
        return -1;
    }
}
else if(c4x < c3x and (closestcorner == 3 or closestcorner == 4)){
    xint = c4x;
    while(xint <= c3x){
        xint += 0.01;
        yint = slope1 * xint - (slope1 * c3x) + c3y;
        leftside = ballslope * (xint - ballx);
        rightside = yint - bally;
        if(Math.abs(leftside - rightside) < 0.1){
            //ball rebound physics go here
            return -1;
        }
    }
}
}

if(c1x <= c4x and (closestcorner == 1 or closestcorner == 4)){
    xint = c1x;
    if(slope2 == Number.POSITIVE_INFINITY or slope2 == Number.NEGATIVE_INFINITY){
        if(c1y <= c4y){
            yint = c1y;
            while(yint <= c4y){
                leftside = ballslope * (xint - ballx);
                rightside = yint - bally;
                yint += 0.01;
                if(Math.abs(leftside - rightside) < 0.1){
                    //ball rebound physics go here
                    return -1;
                }
            }
            xint = c4x + 1;  //ensures the program doesn't go to the next redundant part
        }
        else if(c4y < c1y){
            yint = c4y;
            while(yint <= c1y){
                leftside = ballslope * (xint - ballx);
                rightside = yint - bally;
                yint += 0.01;
                if(Math.abs(leftside - rightside) < 0.1){
                    //ball rebound physics go here
                    return -1;
                }
            }
        }
    }
}
else if(c4x < c1x){
    xint = c4x + 1;  //ensures the program doesn't go to the next redundant part
}
while(xint <= c4x){
xint += 0.01;
yint = slope2 * xint - (slope2 * c4x) + c4y;
leftside = ballslope * (xint - ballx);
rightside = yint - bally;
if(Math.abs(leftside - rightside) < 0.1){
    //ball rebound physics go here
    return -1;
}
}

else if(c4x < c1x and (closestcorner == 4 or closestcorner == 1)){
xint = c4x;
while(xint <= c1x){
xint += 0.01;
yint = slope2 * xint - (slope2 * c1x) + c1y;
leftside = ballslope * (xint - ballx);
rightside = yint - bally;
if(Math.abs(leftside - rightside) < 0.1){
    //ball rebound physics go here
    return -1;
}
}
}

//checkballcollision checks if ball has collided with any robots
function checkballcollision(){
    var dist:Number;
    for(k=0;k<5;k++){
        dist = Math.sqrt(Math.pow((ballx-frobotarray[k]._x ), 2) + Math.pow((bally-frobotarray[k]._y), 2));
        if(dist < 20) return k;
        dist = Math.sqrt(Math.pow((ballx-orobotarray[k]._x ), 2) + Math.pow((bally-orobotarray[k]._y), 2));
        if(dist < 20) return k+10;
    }
    return -1;
}

//verifies which robot has the "right" to kick ball
function checkhasball(){
    var dist:Number;
    for(i=0;i<=4;i++){
        dist = Math.sqrt(Math.pow((ballx-frobotarray[i]._x), 2) + Math.pow((bally-frobotarray[i]._y), 2));
        if (dist < 25) return i;
        dist = Math.sqrt(Math.pow((ballx-orobotarray[i]._x), 2) + Math.pow((bally-orobotarray[i]._y), 2));
        if (dist < 25) return i+10;
    }
    return -1;
}

//vision system gives the robots' and ball's coordinates to the client side as input
function visionsystem(){
    var visioninputs:String = "";
    visioninputs = visioninputs + "f1x" + "=" + frobot_1._x + "&";
    visioninputs = visioninputs + "f1y" + "=" + frobot_1._y + "&";
    visioninputs = visioninputs + "f2x" + "=" + frobot_2._x + "&";
    visioninputs = visioninputs + "f2y" + "=" + frobot_2._y + "&";
    //visioninputs = visioninputs + "f3x" + "=" + frobot_3._x + "&";
    //visioninputs = visioninputs + "f3y" + "=" + frobot_3._y + "&";
    //visioninputs = visioninputs + "f4x" + "=" + frobot_4._x + "&";
    //visioninputs = visioninputs + "f4y" + "=" + frobot_4._y + "&";
    //visioninputs = visioninputs + "f5x" + "=" + frobot_5._x + "&";
    //visioninputs = visioninputs + "f5y" + "=" + frobot_5._y + "&";
    //visioninputs = visioninputs + "o1x" + "=" + orobot_1._x + "&";
    //visioninputs = visioninputs + "o1y" + "=" + orobot_1._y + "&";
    //visioninputs = visioninputs + "o2x" + "=" + orobot_2._x + "&";
    //visioninputs = visioninputs + "o2y" + "=" + orobot_2._y + "&";
    //visioninputs = visioninputs + "o3x" + "=" + orobot_3._x + "&";
    //visioninputs = visioninputs + "o3y" + "=" + orobot_3._y + "&";
    //visioninputs = visioninputs + "o4x" + "=" + orobot_4._x + "&";
    //visioninputs = visioninputs + "o4y" + "=" + orobot_4._y + "&";
    //visioninputs = visioninputs + "o5x" + "=" + orobot_5._x + "&";
    //visioninputs = visioninputs + "o5y" + "=" + orobot_5._y + "&";
    return visioninputs;
}
submittedData = new LoadVars();
submittedData.inputData = visioninputs;
Appendix B: PHP Script Code

Source code from “caretaker.php”

```php
<?PHP

$receivedFromFlashData1 = $_POST['inputData'];
$counterFromFlash = $_POST['counter'];
$txtfileArray = @file("blueteam.txt");

if($myTextFileHandler){
    $writeInTxtFile = @fwrite($myTextFileHandler, "$receivedFromFlashData1");
    @fclose($myTextFileHandler);
    $abc = "success";
    echo ";verify=$abc";
} else{
    $abc = "failed";
    echo ";verify=$abc";
    @fclose($myTextFileHandler);
}
?>
```

Source code for “caretaker2.php” and “caretaker3.php” is similar to “caretaker.php” but the filename changes from “blueteam.txt” to “visionsystem.txt”, and “redteam.txt” respectively.