Proceedings of the 2007 IEEE Symposium on
Computational Intelligence and Games (CIG 2007)

Micro Robot Hockey Simulator – Game Engine Design

Wayne Y. Chen
Experimental Robotics Laboratory
School of Engineering Science
Simon Fraser University, Burnaby, BC, Canada
waynec@fas.sfu.ca

Shahram Payandeh
Experimental Robotics Laboratory
School of Engineering Science
Simon Fraser University, Burnaby, BC, Canada
shahram@fas.sfu.ca

Abstract—**Like robot soccer, robot hockey is a game played between two teams of robots. A robot hockey simulator has been created, for the purpose of game strategy testing and result visualization. One major modification in robot hockey is the addition of a puck-shooting mechanism to each robot. As a result, the mechanics of interaction between the robots and the hockey puck become a key design issue. This paper describes the simulator design considerations for robotic hockey games. A potential field-based strategy planner is implemented which is used to develop strategies for moving the robots autonomously. The results of the simulation study show both successful cooperation between robots (on the strategy level), and realistic interaction between robots and the puck.**

Keywords: Game Engine, Robot Hockey, Artificial Potential Field

I. Introduction

One of the many goals of a robot soccer system is to create a test bed such that many notions from robotics, AI, software engineering, electronics, and mechanics can be synthesized in order to solve a very complex problem. Aside from research challenges, robot soccer also has great entertainment and education values. Ultimately, the study and developments of robot soccer system (multiple corporative agents) can be applied to fields such as search and rescue, service robotics, security, and space missions.

Due to the popularity of the hockey sports, a separate new league for robot hockey is created by the University. We believe that the addition of robot hockey league will help further research activities and public interests in the area of robotics.

The novelty of robot hockey lies in its new robot mechanical design, puck and rink design (e.g. using an air hockey table as the game arena), new rules, and interface for manual robot control [1]. Each hockey robot is equipped with a shooting mechanism that mimics a hockey player's *slap-shooting* the puck (Fig. 1). A miniature puck is used, and the puck floats on the surface of the hockey rink. As in real sports, robot hockey features different rules than robot soccer (e.g. allowing *body check*). In addition to autonomous control, the users can choose to manually control one robot, in cooperation with other computer-controlled teammate robots. These differences give rise to new challenges in robot control and coordination.

To test robot team strategy, it is desirable to have a computer simulation program that runs teams of robots based to the strategies provided by robot programmers. For realistic result visualization purposes, the simulation program should also include graphical representations of the robot hockey environment and the simulated physics that handle interactions among objects in the scene.

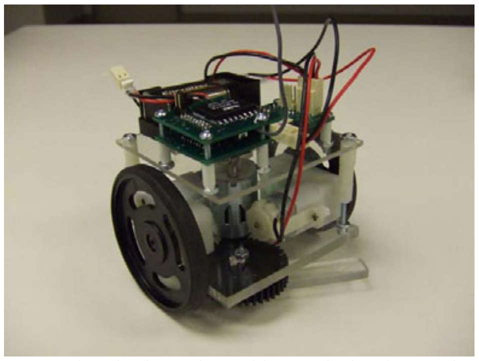
Fig. 1. A micro hockey robot, with a puck-shooting mechanism attached to the bottom-front of its frame

In robot soccer, competitions are held not only between real robots, but also between simulated robots. Competition participants write their own soccer-playing strategies using simulator software provided by the competition organizers. During competitions, the two team's strategies are run on two separate simulator clients, and the simulator server displays the simulated robot soccer game. The *RoboCup Soccer Simulator* is the official simulation software for the Simulation Leagues run by RoboCup [2]. This simulation software is freely available and it supports two teams of 11 autonomous robots. The Federation of International Robot-soccer Association (FIRA) [3] also holds competitions in simulated robot soccer – the *SimuroSot*. Simulation software is provided to participants to write their own team strategies, and the simulator supports 5-on-5 or 11-on-11 games.

As for simulator software for mobile robots in general, the 2D simulator *Stage* [4] and 3D simulator *Gazebo* [5] are freely available robot simulation software that not only allows programmers to test their robot control algorithms through simulation, but also to transfer the control programs to real robots. In simulation, each robot is abstracted as a "Player", with customizable sensors and actuators. The integration of simulation and hardware programming speeds up robot prototyping and development time significantly. Another mobile robot simulation software is the *Webots*. It is developed by Cyberbotics [6] and is commercially available. Webots provides users an integrated development environment for robot modeling and programming. It also features simulation movie recording and transferring of control programs to real robots.

Our eventual objective is to create a robot simulator that not only specializes in the sports of hockey, but it is also expandable to provide desirable features such as simulation competition (as in RoboCup and FIRA Simulators) and code-transfer to real hardware (as in Stage, Gazebo, and Webots). In addition, the simulator should support joystick inputs for manual robot control (as in typical video games).

This paper is organized as follows: Section II describes the overall architecture design of the simulator. Sections III gives more detailed design issues and proposed solutions relevant to the hockey game. And the following major aspects of the simulator are discussed: rendering, strategy, inverse kinematics, motion scheduling, forward kinematics, and event dynamics. Section IV demonstrates simulator performance through test runs with single or multiple robots, and with or without the puck. Section V concludes the paper and suggests future work.

II. Game Engine Architectural Design

The architecture of the simulator engine follows an object-oriented design, which makes it easy to maintain, modify, and reuse. The current simulator consists of nine major components: current state, rendering, forward kinematics, inverse kinematics, event dynamics, motion scheduler, strategy, manual control input, and the graphical user interface. Fig. 2 illustrates the architectural block diagram of the design.

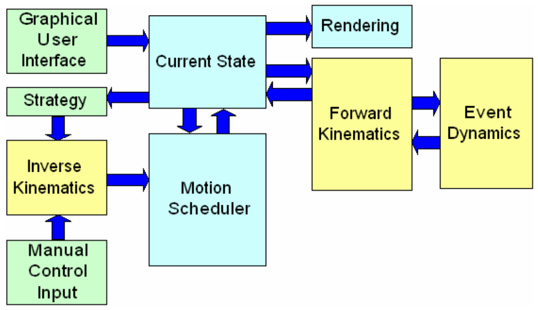

Fig. 2. Architectural design of the robot hockey simulator.

The **current state** module stores all state variables at each time stamp of simulation. This module manages information such as positions, velocities, and accelerations of the puck and all robots, as well as environment properties such as time and surface property of the hockey rink.

The **graphical user interface** includes controls that allow users to view or set the current state, change the camera angle, and pause or change the speed of simulation.

The **strategy** module decides the actions for each robot, based on information from current state. Robot actions are expressed in terms of high-level commands such as navigating to certain waypoint, shooting the puck, passing the puck, etc.

The **manual control input** module maps the user's input from the keyboard or joystick into robot actions. The users have the option of controlling one of the robots manually, and the users' commands have higher priority over those generated by the strategy module. Manual control provides a second way for robot testing in simulated environment. The *hybrid* control model, in which human and computer cooperate to run a team, is also an interesting research topic.

The **inverse kinematics** module converts high-level commands, such as navigating to certain waypoint, into low-level robot control commands, such as turning the wheels at certain velocities for certain time duration.

The **motion scheduler** keeps track of a list of low-level robot commands that need to be executed by the end of the current rendering frame. Each robot action is associated with a time stamp. When the global clock of the simulator reaches this time stamp, the expired command is sent to the current state module and removed from the motion scheduler.

The **forward kinematics** module calculates the robots' and the puck's states for the next rendering frame, based on information from current state. The calculated new states need to be verified by the event dynamics module, before they can be sent back to the current state module for update.

The **event dynamics** module checks if the new states from forward kinematics calculations would cause collisions between objects. If collision should occur, collision response calculations are performed to modify the new states. After the new states are modified and verified, they are sent back to the forward kinematics module.

The **rendering** module reads the position and orientation information of the robots and the puck from current state, and draws them in the scene. This module specifies the 3D models of all objects in the scene: the robots, the puck, and the hockey rink.

In summary, at the beginning of each rendering frame, the rendering module draws the robots, the puck, and the hockey rink in the scene based on the current state. Then the strategy module decides the next actions of the robots, based on the current state. If there are user inputs from keyboards or joysticks, the manual control input module replaces the strategically generated robot actions with the user specified ones. The inverse kinematics module converts the high-level actions into low-level robot commands. The low-level commands are then sent to the motion scheduler. The motion scheduler checks the current time, and picks from its list the robot commands whose time stamp expires. The selected commands are sent to the current state module for state update. The forward kinematics module calculates the new states of the robots and the puck based on the updated current state information. The event dynamics module checks for collision and modifies the calculations from the forward kinematics module if there is collision. The verified new states are passed back to the forward kinematics module, which then send these new states to the current state module for update. The simulator advances to the next rendering frame by incrementing its global clock, and the whole process repeats. Fig. 3 summarizes the process flow.

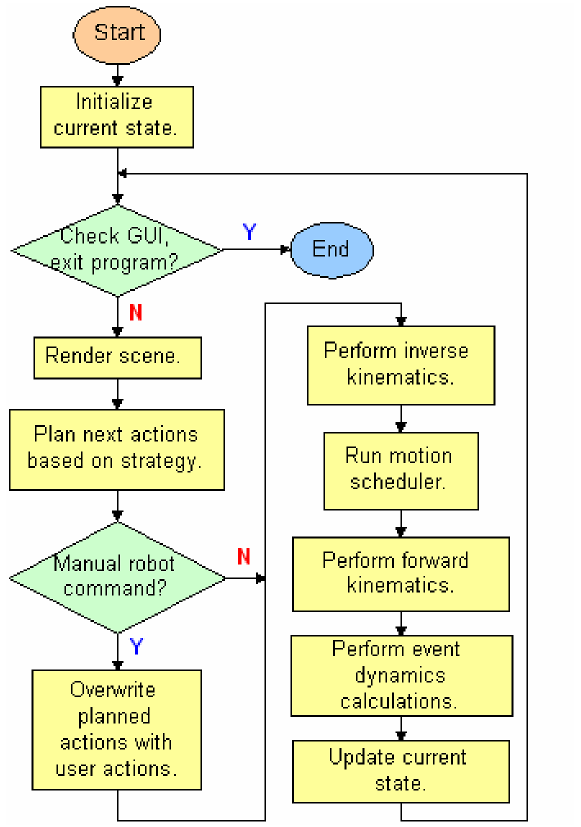
Start
Initialize current state.
Check GUI, exit program?
Y
End
N
Render scene.
Plan next actions based on strategy.
Manual robot command?
N
Y
Overwrite planned actions with user actions.
Perform inverse kinematics.
Run motion scheduler.
Perform forward kinematics.
Perform event dynamics calculations.
Update current state.

Fig. 3. Logic flow of each rendering frame in simulation.

III. Design Considerations due to Robot Hockey

This section gives an overview of various components of our proposed simulator. The following major components of the game engine are discussed: rendering, strategy, motion scheduler, kinematics, and event dynamics.

A. Rendering

OpenGL is used to create the 3D scene, which consists of the hockey rink, a hockey puck, and six hockey robots (two on each team).

The hockey rink has two goal posts, one on each end of the floor. When a robot scores a goal, the light bulb on top of the goal lights up. Similar to real hockey, the rink can be modified to allow robots and the puck to travel behind the goals. Fig. 4 illustrates the overhead view of the hockey rink model.

Each hockey robot is modeled as a cube, plus two wheels and a kicker. On each wheel, we draw a white circle as a marker, so that the wheel's turning motion is easily visible. The kicker is modeled as a rectangular rod that pivots to the front-left corner of the robot. For robot identification, each robot has unique textures on its top (color identifiers) and on its front/back (name tags). The users can customize these textural images to their own liking. Fig. 5 illustrates models of two hockey robots and a puck.

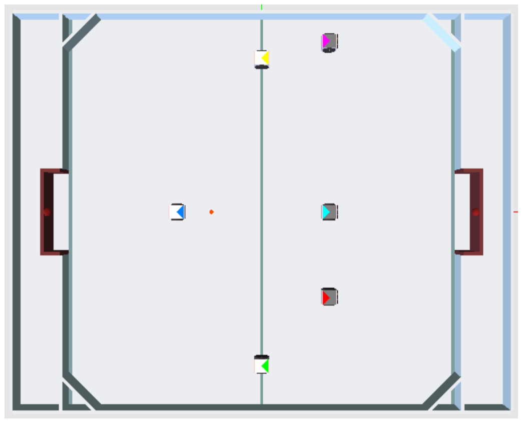

Fig. 4. Model of the hockey rink (overhead view).

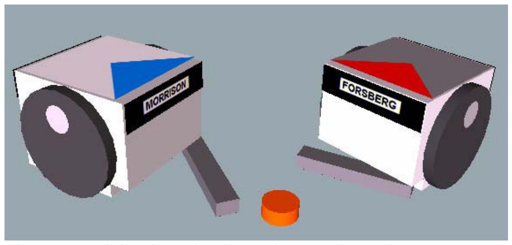

Fig. 5. Models of two hockey robots and a puck.

B. Strategy

To test the functionality of the simulator, we have implemented a potential field based strategy planner [1][7].

We define various potential functions, each of which models a specific game condition. Like a contour map, the robot is to move from locations with high potential to those with low potential. Here, we describe some of the important potential fields used in the game of robot hockey: the base field, distance-to-destination field, play-zone field, line-of-sight field, and robot-personal-region field.

The **base field** enables a robot to move toward the opponent's goal. The base field potential function is directly proportional to the distance from the opponent goal:

$$U_{base} = \gamma_{base} \cdot \sqrt{\left(x - x_{opp\_goal}\right)^2 + \left(y - y_{opp\_goal}\right)^2}, \quad (1)$$

where γ_{base} is a tunable scaling factor.

The **distance-to-destination field** enables a robot to go to certain location. The potential function for this field is directly proportional to the distance from the destination:

$$U_{dest} = \gamma_{dest} \cdot \sqrt{\left(x - x_{dest}\right)^2 + \left(y - y_{dest}\right)^2}. \quad (2)$$

The **play-zone field** enables a robot to stay within its zone of responsibility. The potential values for regions in the robot's play-zone are zero, and a large constant elsewhere. Assuming that the rink has width l_{rink_w}, the play-zone potential function for a robot playing *center* is defined as:

$$U_{zone} = \begin{cases} 0 & y \in \left[-l_{rink\_w}/4, l_{rink\_w}/4\right] \\ \gamma_{zone} & \text{elsewhere} \end{cases}. \quad (3)$$

The **line-of-sight field** enables a robot to stay in regions where it can maintain line-of-sight to the puck. Regions behind the opponent robots facing away from the puck are given high constant potential values:

$$U_{sight} = \begin{cases} \gamma_{sight} & \text{behind opponent w.r.t. puck} \\ 0 & \text{elsewhere} \end{cases}. \quad (4)$$

The **robot-personal-region field** *discourages* a robot from going near the immediate regions near other robots (for collision avoidance). Regions within certain radius $d_{personal}$ from any robots are given high constant potential values:

$$U_{personal} = \begin{cases} \gamma_{personal} & \text{robot clearance} \le d_{personal} \\ 0 & \text{elsewhere} \end{cases}. \quad (5)$$

When calculating the best waypoints for offense, defense, goal-scoring, or goal-tending, different sets of potential functions are added together. The location corresponding to the lowest potential in the super-imposed potential field is the best waypoint for the robot to move to. For visualization, Fig. 6 illustrates super-positioning the five fore-mentioned potential functions to find the best "offense waypoint" (at the cross mark).

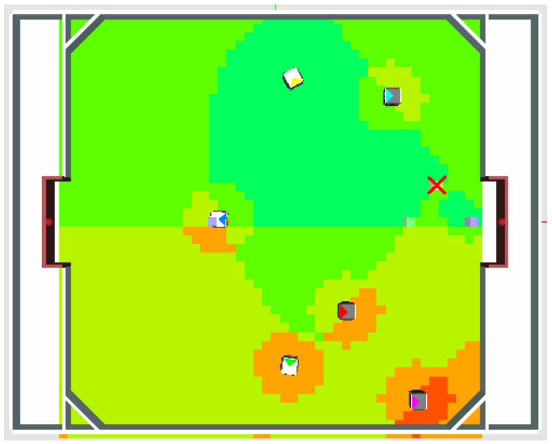
Fig. 6. Finding the best offense waypoint through potential function super-position.

When the team is on offense, the robot that is closest to the puck is selected to be the puck-handler. In the current implementation, the puck-handler chooses its action *randomly* among "shoot to goal", "pass to teammate", and "move with the puck" (i.e. dribble the puck). The other non-puck-handler robots move to their own offense waypoints.

For each robot, potential function super-position is performed to find the best waypoints to perform the selected actions (goal-scoring, passing, dribbling, or moving to offense waypoint). These waypoint locations, along with the information whether or not to activate the kicker, form the high-level action commands. These high-level actions are then sent to the inverse kinematics module for conversion into low-level robot commands.

C. *Inverse Kinematics*

Inverse kinematics calculations deal with the problem that given the desired final configuration of a robot (position and orientation), find the robot joint angles to achieve such configuration. Calculations to be performed in inverse kinematics depend on the mechanical construct of the robots. For the hockey robot shown in Fig. 1, inverse kinematics involves the calculations of three joint angles: left wheel, right wheel, and the kicker.

Here, we show how low-level commands are calculated from high-level commands "navigating to certain waypoint" and "activating the kicker".

To **navigate a robot to a waypoint** at P_f from its current position P_i, we first break the robot's straight path into segments, and each segment has length of the virtual sensing range of the robot. This decomposition is needed because a robot may take more than one rendering frame to reach its destination. Fig. 7 illustrates the decomposition process.

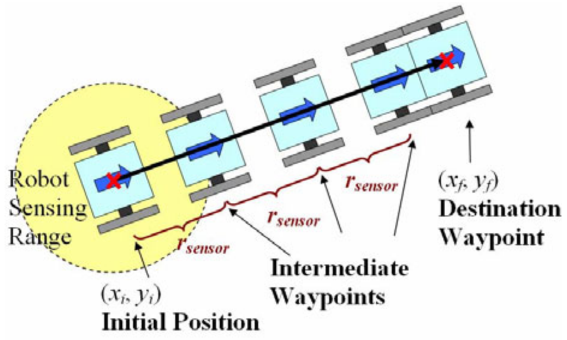

Fig. 7. Decomposition of a robot's path based on its sensing range.

If other robots are located inside the robot sensing range, the *first* intermediate waypoint is adjusted 90 degrees to the left or to the right, as shown in Fig. 8.

After an intermediate waypoint is selected as destination, we calculate the corresponding left and right robot wheel motions to reach this destination. One navigation movement involves the robot first to turn and face the destination, and then move forward toward the destination (as shown in Fig. 9). The angle that the robot needs to turn ($\theta_{turning}$) can be calculated from the current robot position (x_i, y_i), current robot orientation θ_{body}, and the destination position (x_f, y_f):

$$\theta_{turning} = \arctan\left(\left(y_f - y_i\right)/\left(x_f - x_i\right)\right) - \theta_{body}. \quad (6)$$

For a two-wheel robot, the general angular velocity of the robot (ω_{robot}) can be calculated from the linear velocities of the two wheels (v_L, v_R) and the width of the robot (l_{robot_width}), as follows [8]:

$$\omega_{robot} = \left(v_R - v_L\right)/l_{robot\_width}. \quad (7)$$

Assume the wheel motor has maximum angular speed v_{MAX}, and the wheel has radius r_{wheel}. Then the maximum linear speed of the wheel is:

$$\max\left(v_{wheel}\right) = \omega_{MAX} \cdot r_{wheel}. \quad (8)$$

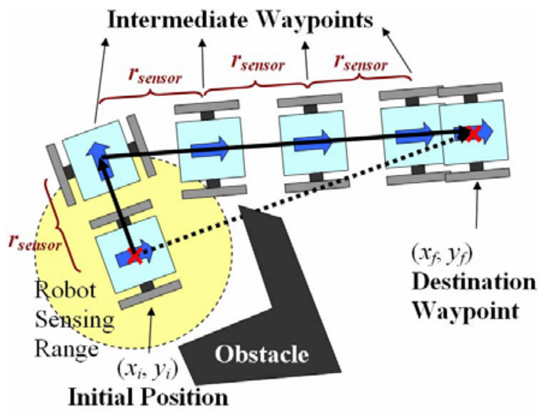

Fig. 8. Modifying the intermediate waypoints due to obstacles.

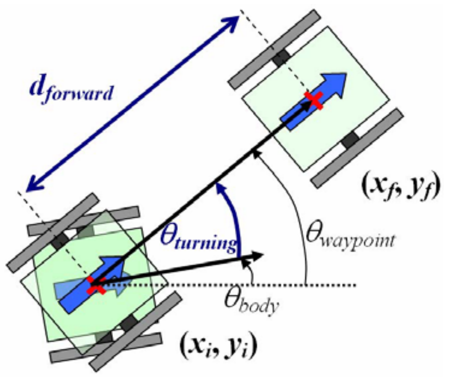

Fig. 9. Robot turning and moving forward to reach destination.

And the maximum angular speed the robot can rotate is:

$$\max\left(\omega_{robot}\right) = 2 \cdot \omega_{MAX} \cdot r_{wheel} / l_{robot\_width} . \tag{9}$$

Suppose that the robot runs its wheels at maximum speed while adjusting its orientation toward the waypoint. Using the results from (6) and (9), the time it takes for the turning motion is:

$$t_{turning} = \theta_{turning} / \left(\max\left(\omega_{robot}\right)\right) . \tag{10}$$

The distance the robot needs to move forward to reach the destination waypoint is:

$$d_{forward} = \sqrt{\left(x_f - x_i\right)^2 + \left(y_f - y_i\right)^2} . \tag{11}$$

Running the two wheel motors at top linear speeds as in (8), the time it takes to complete the forward motion is:

$$t_{forward} = d_{forward} / \left(\max\left(v_{wheel}\right)\right) . \tag{12}$$

In the current implementation, the low-level commands (called **state update vectors**) that get sent to the motion scheduler consist of the following fields: robot ID, robot part, part angular velocity (ω_{part}), part angular acceleration (α_{part}), and start time (t_{start}). This command data structure is chosen because it resembles the commands used to control real hockey robots [8].

The state update vector that commands robot *i* to first turn left, and then move forward toward the destination waypoint looks like those in Table 1. In Table 1, $t_{current}$ is the current time, which refers to the global clock during the current rendering frame in simulation. And the last two rows in the table commands the robot to stop moving after it has finished turning and moving forward.

TABLE I
ROBOT COMMANDS FOR WAYPOINT NAVIGATION

Robot ID	Part Name	ω_{part}	α_{part}	t_{start}
i	Left wheel	$-\max(\omega_{wheel})$	0	$t_{current}$
i	Right wheel	$\max(\omega_{wheel})$	0	$t_{current}$
i	Left wheel	$\max(\omega_{wheel})$	0	$t_{current}+t_{turning}$
i	Right wheel	$\max(\omega_{wheel})$	0	$t_{current}+t_{turning}$
i	Left wheel	0	0	$t_{current}+t_{turning}+t_{forward}$
i	Right wheel	0	0	$t_{current}+t_{turning}+t_{forward}$

When performing a **kicking action**, the robot rotates its kicker out to an angle θ_{kick}, and then retracts the kicker back to its home position. Let $\max(\omega_{kicker})$ be the maximum angular speed of the kicker motor. The time the kicker needs to turn in each direction is:

$$t_{kicking} = \theta_{kick} / \left(\max\left(\omega_{kicker}\right)\right) . \tag{13}$$

If robot *i* needs to perform the kicking action at the end of its waypoint navigation, we append the entries in Table 2 to the end of Table 1. For this case, the time stamp at the end of kicking action should be:

$$t_{kick\_end} = t_{current} + t_{turning} + t_{forward} + t_{kicking} . \tag{14}$$

D. *Motion Scheduler*

The motion scheduler upkeeps a table that holds all the state update vectors corresponding to the commands generated by the strategy and the user control input modules.

TABLE II
ROBOT COMMANDS FOR KICKER ACTIVATION

Robot ID	Part Name	ω_{part}	α_{part}	t_{start}
i	Kicker	$\max(\omega_{kicker})$	0	$t_{current}+t_{turning}+t_{forward}$
i	Kicker	$-\max(\omega_{kicker})$	0	$t_{current}+t_{turning}+t_{forward}$
i	Kicker	0	0	$t_{current}+t_{turning}+t_{forward}+t_{kicking}$

Each state update vector needs to be executed, at the time according to its time stamp. The motion scheduler is responsible for checking the current time (from the current state module), going through the entire list of state update vectors, and picking out the state update vectors whose time stamps have expired. The selected state update vectors are sent to the current state module for current state update, and removed from the scheduler's list.

One major advantage of using the state update vectors and the motion scheduler is that the simulator is able to move multiple robots concurrently and independently. Another advantage of using such design is that the simulator can easily be extended to control real hockey robots. The state update vectors contain low-level motor control information, which can be sent directly to the robots as commands.

E. *Forward Kinematics*

Forward kinematics calculations involve the determination of a robot's configuration from the robot's joint angles. In robot hockey, we perform forward kinematics to calculate a robot's position and orientation for the next rendering frame, based on the current states of its two wheels.

For a two-wheel robot, when the two wheels' velocities are not equal, the robot rotates about an instantaneous center of rotation (ICR), as shown in Fig. 10.

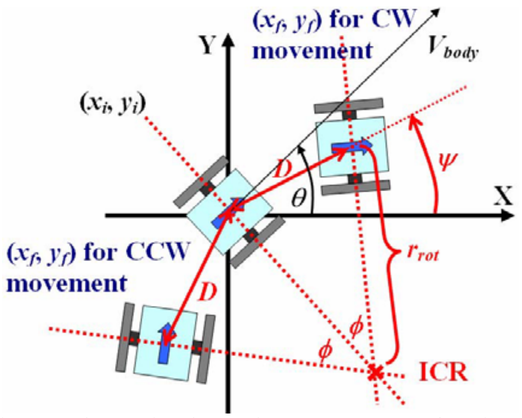

Fig. 10. Robot rotating about an instantaneous center of rotation.

The general angular velocity of the robot (ω_{robot}) can be calculated using (7). Given that the time increment between frames in simulation is Δt, a robot's angle of rotation per frame can be calculated as:

$$\phi = \omega_{robot} \cdot \Delta t \,. \tag{15}$$

The radius of rotation r_{rot} can be calculated from the linear speeds of the two wheels (v_L, v_R), and the robot's body width (l_{robot_width}) as follows [8]:

$$r_{rot} = \frac{l_{robot\_width}}{2} \cdot \frac{v_R - v_L}{v_R + v_L} \,. \tag{16}$$

From r_{rot} and ϕ, we can calculate the magnitude of the robot's displacement due to its rotation around the ICR:

$$D = 2 \cdot r_{rot} \cdot \sin\left(\phi/2\right) . \tag{17}$$

Using the geometric setup as shown in Fig. 10, we can calculate the angle of the robot's displacement (ψ). For the case that the robot moves in the clockwise direction, we use:

$$\psi = \theta - \left(\frac{\pi}{2} - \frac{\pi - \phi}{2}\right) = \theta - \frac{\phi}{2} \,. \tag{18}$$

For the case that the robot moves in the counter-clockwise direction, we use:

$$\psi = \theta + \pi + \left(\frac{\pi}{2} - \frac{\pi - \phi}{2}\right) = \theta + \pi + \frac{\phi}{2} \,. \tag{19}$$

Finally, the location of the robot for the next time frame (x_f, y_f) can be determined from D and ψ, using trigonometry:

$$\begin{aligned} x_f &= x_i + D \cdot \cos\psi \\ y_f &= y_i + D \cdot \sin\psi \end{aligned} \,. \tag{20}$$

F. *Event Dynamics*

Event dynamics calculations deal with changes in motion due to interaction among objects. In simulated environment, this includes collision detection and collision response.

The first step of collision detection is to choose a geometric model to represent the objects. Then by perform intersection check between the geometries, we can determine if two objects collide or not.

The robot hockey environment consists of the hockey rink, the puck, and the robots. For **collision detection**, the hockey rink is modeled as a collection of line segments. The puck is modeled as a line segment, with its two endpoints being the puck's current position and the predicted position for the next rendering frame. The robot's body is modeled as a rectangle. As for the robot's kicker, it uses a line segment model when both the robot's body and its kicker are stationary. When the robot's kicker is stationary but its body is moving, the kicker is modeled as a parallelogram. When the robot's body is stationary but its kicker is swinging, the kicker is modeled as a pie shape. Fig. 11 illustrates the three geometric models of the robot's kicker.

In robot hockey, the objective is for a team of robots to manipulate the puck with their bodies and kickers, until the puck reaches inside the opponent's goal. Thus, **collision response** of the puck plays a significant role in both the game play and game physics.

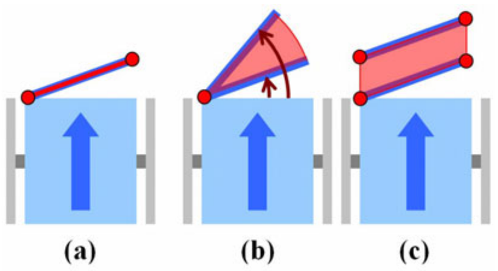

Fig. 11. Geometric models of the kicker for collision detection: (a) a line segment, (b) a pie, (c) a parallelogram.

Suppose that collision detection finds out that the puck will collide with a surface at location P_C. Fig. 12 illustrates the scenario, where V_i is the puck's velocity vector before impact, and V_f is the puck's velocity after impact.

Given that the normal vector of the contact surface is N_C, we can break down the puck's initial velocity into two components: one in the normal direction $V_{N,i}$ (relative to N_C), and one in the tangential direction $V_{T,i}$, by using dot product and vector subtraction:

$$\begin{aligned} V_{N,i} &= \left(V_i \bullet N_C\right) N_C \\ V_{T,i} &= V_i - V_{N,i} \end{aligned} \quad . \tag{21}$$

By applying the conservation of momentum, the puck's speed in the normal direction after impact can be calculated using the following simplified equation (where ε is the coefficient of restitution):

$$V_{N,f} = -\varepsilon \cdot V_{N,i} \,. \tag{22}$$

The puck's speed in the tangential direction relative to the surface normal remains unchanged:

$$V_{T,f} = V_{T,i} \,. \tag{23}$$

The overall velocity of the puck after impact is then the sum of the normal (22) and the tangential (23) components.

If the surface of collision also has velocity $V_{surface}$ at the time of impact, the surface velocity contributes to the puck's final velocity as well:

$$V_f = V_{N,f} + V_{T,f} + V_{surface} \,. \tag{24}$$

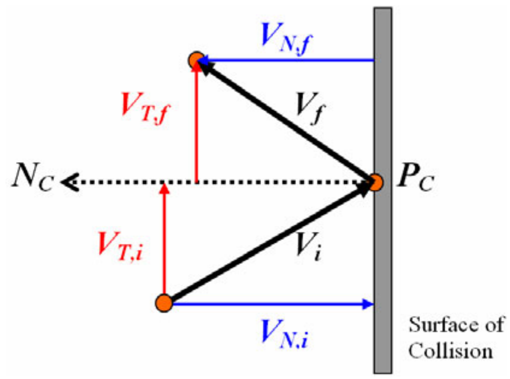

Fig. 12. Impact analysis of the puck bouncing off a surface.

When a robot's kicker strikes the puck at location P_C (Fig. 13), the kicker's velocity at the point of contact can be calculated from the angular velocity of the kicker ω_{kicker} and distance from the kicker's hinge P_{hinge} to contact point P_C, as follows:

$$V_{kicker} = \omega_{kicker} \cdot \mathrm{dist}\left(P_{hinge}, P_C\right) . \tag{25}$$

If the robot swings its kicker at the puck while moving forward at the same time with linear velocity V_{body}, then the overall surface velocity that contributes to the puck's final velocity after impact is:

$$V_{surface} = V_{kicker} + V_{body} \,. \tag{26}$$

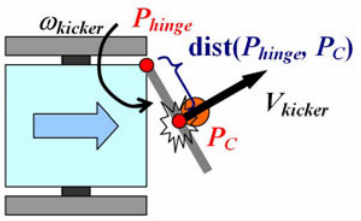

Fig. 13. The robot shoots the puck by swinging its kicker.

When the puck slides freely in the rink, it is subject to the kinetic friction between the puck and the rink surface. The puck's deceleration due to this frictional force is:

$$\partial V_{puck} / \partial t = -\mu_k \cdot g \cdot \left(V_{puck} / \left\| V_{puck} \right\| \right), \tag{27}$$

where μ_k is the coefficient of kinetic friction and g is the gravitation acceleration ($g = 9.81$ m/sec^2).

IV. PERFORMANCE OF THE SIMULATOR

This section demonstrates the performance of some main features of the robot hockey simulator.

The GUI of the simulator is implemented using Glui [9]. The GUI includes controls to pause/resume animation, to view/edit all current states, to choose strategies of the two teams, and to rotate/pan/zoom the global camera. Fig. 14 shows the current GUI design.

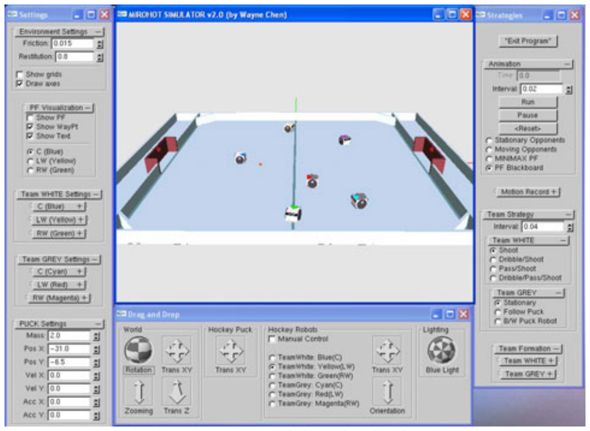
Friction: 0.015
Restitution: 0.8
[✓ Draw axes]
[✓ Show WayPt]
[✓ Show Text]
[✓ C (Blue)]
Mass: 2.0
Pos X: -31.0
Pos Y: -6.5
Vel X: 0.0
Vel Y: 0.0
Acc X: 0.0
Acc Y: 0.0
Time: 0.0
Interval: 0.02
[✓ PF Blackboard]
Interval: 0.04
[✓ Shoot]
[✓ Stationary]
[✓ TeamWhite: Yellow(LW)]

Fig. 14. Graphical user interface of the robot hockey simulator.

Fig. 15 shows an isometric view of the playfield, with potential fields in the background. On average, the simulator runs at 20 frames per second (CPU: Intel® Pentium® 1.5GHz; RAM: 512MB; OS: Windows® XP). The maximum frame rate can go as high as 25 frames per second. Strategy calculations are performed every other frame (10 frames per second).

Some basic robot hockey skills are waypoint navigation, puck shooting, and puck passing. Fig. 16 illustrates a robot navigating to a waypoint (to the cross mark), while avoiding an obstacle in its away. Fig. 17 illustrates a robot shooting the puck with its kicker: (a) with only its kicker swinging, (b) with both its kicker swinging and its body moving forward. Fig. 18 illustrates robot cooperation: a robot passing the puck to another teammate robot.

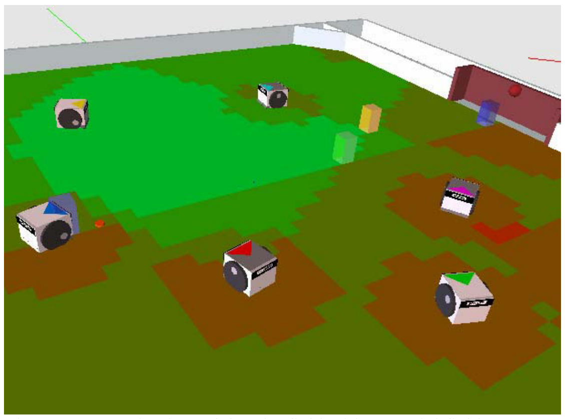
Fig. 15. Isometric view of the playfield with potential field.

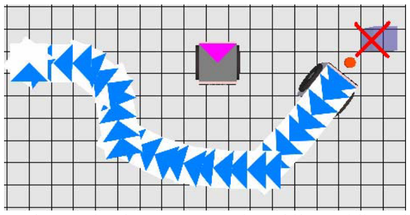
Fig. 16. Robot navigation to a waypoint while avoiding obstacles.

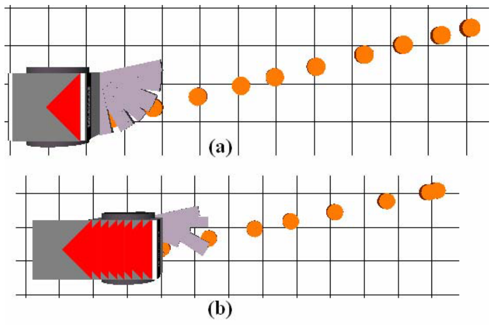

Fig. 17. Shooting the puck: (a) with kicker, (b) with kicker and body.

As discussed in Section III-D, the use of state update vectors and the motion scheduler enables the simulator to animate multiple robots concurrently and independently. Fig. 19 illustrates two teams of robots (three robots on each team) competing in a game of robot hockey.

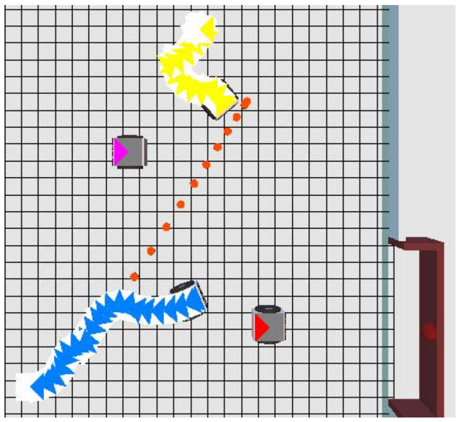
Fig. 18. Passing the puck to a teammate.

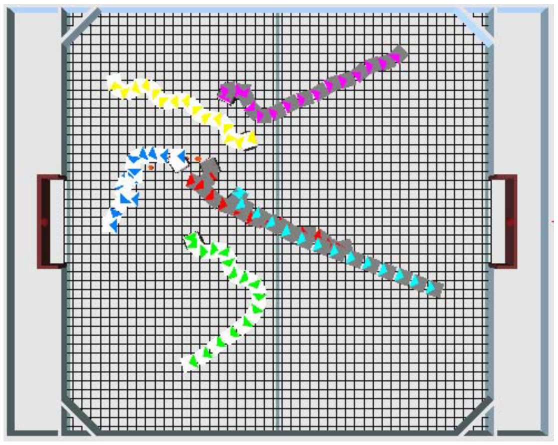
Fig. 19. Two teams of robots competing in robot hockey.

V. Conclusion and Future Work

This paper describes the game engine design of a robot hockey simulator, both on the architectural level and on the component level. Future work includes robot dynamics model, improved puck kinematics model (spinning effect), improved team strategies, larger robot teams, a programming interface for strategy and robot model design, transfer of control codes to real robots, and simulated robot hockey tournaments over the Internet.

Acknowledgment

The authors would like to thank Shahrad Payandeh for his suggestions and support to the robot hockey project.

References

[1] W. Y. Chen and S. Payandeh, "Passer-receiver coordination under multiple defenders in the game of robot hockey," in *Proceedings of the 2005 FIRA RoboWorld Congress*.

[2] Official Website of *RoboCup* [Online]. Available: http://www.robocup.org.

[3] Official Website of the *Federation of International Robot-soccer Association* [Online]. Available: http://www.fira.net.

[4] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage Project: tools for multi-robot and distributed sensor systems," in *Proceedings of the 2003 IEEE International Conference on Advanced Robotics*, pp. 317-323.

[5] N. Koenig and A. Howard, "Design and use paradigms for Gazebo: an open-source multi-robot simulator," in *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2149-2154.

[6] Business Website of *Cyberbotics* [Online]. Available: http://www.cyberbotics.com.

[7] P. Vadakkepat, T. H. Lee, and L. Xin, "Application of evolutionary artificial potential field in robot soccer system," in *Proceedings of the 2001 IFSA World Congress and the 20th NAFIPS International Conference*, pp. 2781-2785.

[8] J. H. Kim, "Lecture notes on EE006 robot soccer system," Korea Advanced Institute of Science and Technology, Taejon, Korea, 1998, pp. 31-40.

[9] P. Rademacher, "Glui – a GLUT based user interface library," University of Northern California, CA, 1999.