Intelligent Three-Dimensional Ultimate Frisbee Simulator

by

David Bailey

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

© David Bailey, MMIII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 21, 2003

Certified by

Seth Teller
Associate Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Intelligent Three-Dimensional Ultimate Frisbee Simulator

by

David Bailey

Submitted to the Department of Electrical Engineering and Computer Science on May 21, 2003, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

This document describes the design and implementation of an intelligent three-dimensional simulator for the sport of Ultimate Frisbee. The simulator is designed to be used as a learning tool and coaching aid, empowering users to learn about the rules as well as the basic strategic concepts of the game. Players make intelligent decisions on the fly instead of executing a fixed routine, and the outcomes of players’ decisions vary nondeterministically based on individual player skills and physical attributes. The simulator is based on an existing Ultimate Frisbee simulator [1], and also incorporates some of the existing simulator’s source code.

Thesis Supervisor: Seth Teller
Title: Associate Professor
Acknowledgments

First and foremost, I want to thank Seth Teller for all his help, for his love for Ultimate Frisbee, and for making it possible for Ultimate Frisbee to be the focus of my Master’s Thesis. I could not have hoped for a more enjoyable topic.

Thanks to Dean Bolton for leading me to the opportunity to work on this project, and for his work on the original Three-Dimensional Ultimate Coaching Simulator [1]. Thanks also to Emery Lin, my thesis project partner, for his hard work [2] and for putting up with me when I disappeared for days at a time for Ultimate tournaments.

Extra special thanks to Mom, Dad, Melissa, and all my friends for every time they helped to make five years at MIT a little bit more bearable.
# Contents

1 Introduction  

2 The Sport of Ultimate Frisbee  

3 Goals  
3.1 Audience  
3.2 Prior Work  
3.3 Project Scope  

4 Achievements  
4.1 Existing Features  
4.1.1 Architecture  
4.1.2 Artificial Intelligence  
4.1.3 Graphical User Interface  
4.2 Improvements  
4.2.1 Game Architecture  
4.2.2 Functionality  
4.2.3 Graphical User Interface  
4.3 Individual Contribution  

5 Implementation  
5.1 Framework  
5.1.1 Isolating the AI  
5.1.2 Collision Prevention  

13  
15  
19  
21  
21  
22  
22  
24  
24  
26  
27  
29  
33  
33  
34  
34
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>An illustration of basic strategy terminology</td>
<td>16</td>
</tr>
<tr>
<td>4-1</td>
<td>The old simulator's 3D display, overhead view and score board</td>
<td>23</td>
</tr>
<tr>
<td>4-2</td>
<td>The old simulator's simulation and camera controls</td>
<td>24</td>
</tr>
<tr>
<td>4-3</td>
<td>The new simulator's 3D display</td>
<td>30</td>
</tr>
<tr>
<td>4-4</td>
<td>The new simulator's overhead view and score board</td>
<td>31</td>
</tr>
<tr>
<td>5-1</td>
<td>The division between the rules and physics component and the AI component</td>
<td>33</td>
</tr>
<tr>
<td>5-2</td>
<td>The effects of collision prevention</td>
<td>35</td>
</tr>
<tr>
<td>5-3</td>
<td>The forces that act on a flying frisbee as modeled in this simulation</td>
<td>36</td>
</tr>
<tr>
<td>5-4</td>
<td>The new user interface display</td>
<td>42</td>
</tr>
<tr>
<td>5-5</td>
<td>The score board display when a turnover occurs</td>
<td>43</td>
</tr>
<tr>
<td>5-6</td>
<td>The new user interface controls</td>
<td>43</td>
</tr>
</tbody>
</table>
List of Tables

5.1 The characteristics of different throw types . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
Chapter 1

Introduction

This paper describes the design and implementation of a realistic three-dimensional Ultimate Frisbee simulator that can be used as a learning tool and a coaching aid. Users of the simulation will be able to see how the rules of the game determine the course of play, as well as how the basic strategies are carried out by the offense and defense.

The simulator uses realistic physics to make disc and player movement look as natural as possible. Players use artificial intelligence to make decisions based on game situations rather than following a predetermined series of actions, and also nondeterministically make errors in executing their actions. Each player has different skill and physical ability ratings, which allows the user to see how differences in player attributes affect the game play, as well as how the game changes when players' attributes are changed.

Prior to the start of this project, one version of an Ultimate Frisbee simulator had already been written [1]. The existing simulator had a graphical user interface, as well as rudimentary physics and artificial intelligence. The Intelligent Three-Dimensional Ultimate Frisbee Simulator uses this existing software simulator as a starting point, and offers several improvements. The new simulator offers a new player model that varies players' abilities based on skill ratings, uses more realistic physics, implements a flexible strategy software component, and offers many improvements to the 3D graphics and user interface.
Chapter 2

The Sport of Ultimate Frisbee

Ultimate Frisbee is a sport played by two teams of seven players, and bears similarities to several other sports such as soccer, football, and basketball. The game is played on an outdoor field slightly smaller than a football field. Players attempt to complete passes with the frisbee to their teammates, with the eventual goal of completing a pass in the opponent’s endzone. Players may not run while they have the disc, and the opposing team gains possession if the offense lets it touch the ground or catches it out of bounds. The first team to score 15 goals wins. The rules of the game are explained in more detail in Appendix A.

In order to understand the discussion of the sport of Ultimate Frisbee that appears in this paper, it is important to understand some terminology and strategy that is basic to the game. Each point starts when the defense pulls, or throws the disc from their endzone to the opposite end of the field. Each defender then usually guards a single opponent. The defender guarding the opponent who has the disc is called the mark, and stands off to one side of him forcing the thrower to throw to one side of the field. This divides the field into the open side, which is the half of the field to which the mark is allowing throws, and the break side, which is the side of the field to which the mark is trying to protect. The remaining defenders then stand between their man and the disc and slightly toward the open side, relying on the mark to prevent throws to the break side, and move to try to intercept or block the disc when it is thrown.

The team on offense usually lines up in a stack, forming a straight line near the
Figure 2-1: An illustration of basic strategy terminology
middle of the field, and players will cut to the open space on either side of the stack. The thrower will then try to throw to one of these players, either by making a throw to the open side, or by trying to break the mark and throw to the break side. Breaking the mark is difficult because it involves throwing past the mark, and can result in a point block if the defender is able to block the throw. Some of the terms defined in this chapter are illustrated in Figure 2.
Chapter 3

Goals

This chapter describes the audience for whom the simulator is intended, followed by an analysis of the scope of prior work and outline of the scope of this simulator. This discussion focuses on goals, rather than actual achievements or implementation details. Achievements and implementation details are discussed in Chapters 4 and 5, respectively.

3.1 Audience

This project is designed to be usable by as wide a range of people as possible. For beginning players, this requires a user interface that is simple to use and makes it easy to understand what is happening in the simulated game. For intermediate players, this requires that the strategies used be sufficiently advanced so that the user still actually learns something from the simulator. Although the simulator's architecture allows for the programming of complex strategies, the program's artificial intelligence attempts implement offensive and defensive strategies that strike a balance between being simple enough for beginning players to understand, yet advanced enough for intermediate players to be able to benefit from. In order for the simulator to be made as widely available as possible, it is written as a Java Applet so that it can be run with a web browser on any computer platform.
3.2 Prior Work

This project is built on the source code of the software Ultimate Frisbee simulator presented in [1]. The new simulator uses this prior work as a starting point, retaining most of the existing features from the first project.

The goal of the existing simulator described in [1] was “to make an Ultimate simulator that could be used as a coaching tool” that was “easy to use as well as easy to distribute to a wide audience.” The existing simulator provides the user with a 3-dimensional display of the players, disc, and playing field in one window. The camera view can be manipulated manually with the mouse to track the action. A second window shows a top view of the field, and a score board with vital game statistics such as the score, possession, and stall count. In addition to being able to watch the game play, the user can also pause, rewind, and replay parts of the simulation. Game play shows players and the disc obeying basic rules of the game, as well as basic strategic principles.

3.3 Project Scope

The goal of this project to was take the simulator described in the previous section, and to make it a more effective learning tool for the viewer. The physics was to become more realistic in order to make the game play appear more natural. The offensive and defensive strategies were to be improved so that the user could gain a basic understanding of the strategies involved from watching the simulator. The user interface and 3-dimensional graphics were also to be improved to make it easier for the viewer to perceive what was happening in the simulated game. Lastly, the software was to be written to be flexible and extensible to facilitate future improvement to the simulator.
Chapter 4

Achievements

Having summarized the goals of both the old and the new versions of the Ultimate Frisbee simulator, the specifics of how each project reached its goals will now be analyzed.

4.1 Existing Features

The features of the existing simulator presented in [1] will be discussed in terms of game architecture, artificial intelligence, and user interface.

4.1.1 Architecture

The existing game architecture implemented a subsumption architecture in which higher priority layers take priority over lower priority layers in determining the action of each player [1]. At the most basic level, players must obey the laws of physics and the rules of the game. Additional layers allow the player to act within the confines of the most basic rules to move around the field and try to help their team win the game. Although the architecture implements multiple layers, there is no software separation between the layers—they are all implemented in the same Java class.
4.1.2 Artificial Intelligence

The artificial intelligence in the original simulator is of approximately the same level as that of beginning Ultimate Frisbee players. Offensive players choose the starting and ending points of their cuts as well as the timing for their cuts at random, rather than using a coordinated strategy. Defensive players trail behind their opponents, without taking advantage of the direction in which the mark is forcing or the direction from the player they are guarding to the disc.

4.1.3 Graphical User Interface

The simulator offers several windows that allow the user to view and control what is happening in the simulation. The main window allows the user to see a 3-dimensional display of the game as it is being played. The overhead view lets the user see a top view of the playing field, along with a score board that provides information about what the score is, who has the disc, and what the stall count is. The simulation control frame offers a convenient interface for starting, stopping, and replaying the simulation, and the camera control frame lets the user change the camera position. Lastly, the player control frame displays information about a particular player if one is selected using the cursor in the playing field window.

3-Dimensional Display

The 3-dimensional display in the main window shows the playing field, both teams, and the disc as the simulation progresses and players and the disc move around the field. The 3-dimensional graphics are powered by Java 3D, and update continuously showing a live visual representation of the game in action. A screen shot from the simulator’s 3-dimensional display is shown in Figure 4-1(a).
Overhead Display

The playing field shows a top view of the entire field, showing the disc and all of the players even when they are out of the view of the camera. This makes it possible to keep track of where the players are on the field if it cannot easily be determined by the camera view. The playing field display also shows a score board, which indicates the score, the current possessing team and player, and the current stall count. An sample overhead view is shown in Figure 4-1(b).

User Controls

The simulation can be controlled from either the simulation control frame, or from the simulation menu in the main window. Either set of controls allows the user to play, pause, rewind and replay the simulation, as well as step forward or backward through the past 30 seconds of simulation at 1-second intervals. The camera view
can be adjusted to one of four fixed positions using the camera control frame, and it
can also be adjusted manually by using a 3-button mouse in the main window where
the graphics are displayed. The user controls are shown in Figure 4-2.

Figure 4-2: The old simulator’s simulation and camera controls

4.2 Improvements

Before any changes to the functionality of the simulator could be implemented, the
game architecture needed to be modified to make it more flexible and extensible.
Once the framework was modified, functionality was added by improving the physics,
adding more variable player attributes, and creating more complex offensive and
defensive artificial intelligence. Once functionality was improved, changes were made
to the graphical user interface to make the program easier to use.

4.2.1 Game Architecture

The graphics and the game play components of the existing simulator were very well
abstracted from each other. However, the physics, rules, and strategy components
within the game play component were all closely entwined, limiting the extent to
which any of these components would be improved. The first step in improving
the simulator was to modify the design of the software without changing any of its
functionality.
Changing the Architecture

The process of modifying the game architecture involved rearranging and reimplementing parts of the original simulator without changing its actual functionality. The existing simulator used a subsumption architecture, in which higher priority layers, such as the rules of the game, took precedence over lower priority layers, such as the player’s decision-making process. However, in the existing simulator, there was no software separation between these layers. The first step in writing the new simulation was to isolate the artificial intelligence from the more basic subsumption layers. This was done by separating out the enforcement of the rules of physics and the rules of the game from each player’s decision making process into separate software components. Isolation of these layers made it easier to change the artificial intelligence without worrying about whether the change would cause the more basic rules to be violated.

Repairing Physics

Some physics functionality had to be reimplemented as a result of isolating the artificial intelligence and physics components. For example, collision prevention was implemented to prevent players from occupying the same space. Although this functionality was already present in the old simulator, it was implemented as part of the offensive and defensive artificial intelligence strategies. The functionality had to be reimplemented so that the players’ new artificial intelligence components would not bear the burden of avoiding collisions.

Repairing Rules of the Game

Some rules of the game also never needed to be enforced by the original simulator because the AI routines made sure never to violate them. For example, passes were thrown to land in bounds under all circumstances, so the disc landing out of bounds was a situation which never had to be dealt with. In order to allow the artificial intelligence layer to operate without such restrictions, functionality had to be added to the game’s framework properly deal with the disc landing out of bounds. This
includes returning the disc to the position at which it first crossed the out of bounds line and, if the disc was caught out of bounds, forcing the player to drop the disc and turning possession over to the other team.

4.2.2 Functionality

Once the game architecture had been improved, changes to the actual functionality of the simulator became possible. These changes included improvements to the physics, player model, and artificial intelligence.

Physics

Other physics functionality was added that did not exist previously. The two types of moving objects in the existing simulator are the players and the frisbee. However, neither of these objects were subject to realistic laws of physics in the existing simulator. Players could move around the field at their maximum speed at any time, even if they were standing still in the previous moment. The disc traveled at a constant horizontal velocity after it was thrown, and dropped at a constant velocity toward the ground. In order to make the game play look more realistic, the new simulator was modified to use realistic physics for both player and disc movement. Players attempting to run at full speed starting from a stationary position now gradually accelerate up to full speed, and forces of lift, gravity and drag are used to compute the frisbee’s path through the air. Player and disc physics were based on data published in outside research papers [3] and [4], respectively, in order to make the simulation look as realistic as possible.

Player Model

The existing simulator provided varying values for each player for a variety of attributes, such as throwing skill, height, and speed. However, many of these attributes did not impact the behavior of the simulator in any way. In order to make game play more realistic and interesting, the software now incorporates these attributes
into the simulation. For example, a player with a low forehand skill level misses his target more often than a player with a higher skill rating, and a player with a higher quickness rating accelerates faster than a player with a lower quickness rating.

Artificial Intelligence

Once the artificial intelligence layer was separated in software from the enforcement of rules of the game and the laws of physics, it became easier to write more advanced offensive and defensive strategies. On offense, players form a stack, and make cuts either toward or away from the disc, depending on where they will be the most open. The thrower, rather than choosing a receiver randomly, evaluates how open they appear to be before making a decision. On defense, players stand on the open side of the field between the player they are guarding and the disc, taking advantage of the fact that the mark prevents throws to the break side. The last man back on defense stands deep of the man he is guarding to keep from being beaten to the endzone on a deep cut. Making the artificial intelligence more complex makes it possible for the user to learn more from using the simulator, making the it more appealing to a wider audience. The details of the artificial intelligence will be explored in more detail in Chapter 5.

4.2.3 Graphical User Interface

One of the goals of our project, after improving the realism and complexity of the game play, was to make it easier for the user to see what was happening in the game. This primarily involved modifications to the user interface. Some of these modifications include adding more components to the overhead view and score board displays, automatically moving the camera view to a useful location, and otherwise improving the appearance of the 3-dimensional graphical display.
Camera View

One problem with the existing 3-dimensional display is that it can be difficult to follow the action as the disc moves to different parts of the field. Although the camera view can be adjusted along three axes by using the mouse buttons in the main window, it is difficult for someone unfamiliar with the simulator to adjust the camera display fast enough to follow the action. The simulator now offers two automatic camera modes: the fixed view, and the thrower view. Both views cause the camera view to rotate to point toward the disc as it moves around the field. With the fixed view, the user can choose between five locations on and around the field from which to view the action. With the thrower view, the camera is positioned behind the most recent thrower, facing downfield toward the endzone the thrower is trying to score in. Making the camera automatically follow the players and disc as they move around the field makes it much easier for the viewer to pay attention to other details of the simulation display, allowing them to concentrate more on the game play and less on moving the camera. Figure 4-3 shows the 3-dimensional display, featuring a view of the field from a point of view just behind the thrower.

Flipped Graphics

Another problem with the old graphics was that due to the choice of x-, y- and z-axes, the 3-dimensional display was a mirrored universe from what was displayed in the 2-dimensional overhead view. The simulator has now been modified to flip the graphics, displaying a 3-dimensional view that is consistent with the 2-dimensional view. This improves the user experience by making it easier for the viewer to correlate the two views and put together where the players and disc are in 3-dimensional space.

Overhead View and Score Board

In the old simulator, the overhead view and score board displayed the basics of what is happening in the game, such as the players’ positions on the field, the score, and which team possesses the disc. However, the display doesn’t lend any help to the
user in terms of understanding what events are taking place on the field and what each team is trying to accomplish. The new simulator attempts to display more information to help the viewer understand which way the offense is traveling, which way the defense is forcing, and when a goal is scored or a turnover occurs. A snapshot of the new overhead view and scoreboard is shown in Figure 4-4.

4.3 Individual Contribution

This project was completed by a team of two—Emery Lin and the author of this paper—working closely together. Included in the following chapter are only the details of the work that the author performed. Those details not included here can be found in [2].

The work done on improving the software simulator included improving the framework, adding functionality, and improving the GUI.

The work done on improving the framework was highly cooperative, but can be roughly fit into the categories of: creating a new API between the rules and physics component and the artificial intelligence component, repairing the enforcement of the rules of the game, and repairing the physics. Repair of the physics implementation is discussed in this paper. Design of the new API as well as repairing the enforcement of the rules of the game are discussed in [2].

Improvements to the functionality of the simulator fall into the categories of physics improvements, strategy improvements, and player model improvements. Defensive strategy, as well as the physics pertaining to the frisbee are discussed in this paper. Player physics, offensive strategy, and improvements to the player model are discussed in [2].

Finally, all changes in the graphics and user interface will be discussed in this paper.
Figure 4-3: The new simulator’s 3D display
Figure 4-4: The new simulator's overhead view and score board
Chapter 5

Implementation

This chapter describes the implementation details completed by the author of this paper. The changes made to the following aspects of the game will be presented in detail here: game framework, disc physics, defensive artificial intelligence, and graphical user interface. All changes which are not detailed in this chapter are discussed in [2].

5.1 Framework

Before any work could be done to improve the AI or physics of the simulator, the two layers had to be separated into different software components.

![Diagram showing the division between the game, team, player, rules and physics module, and AI module](image)

Figure 5-1: The division between the rules and physics component and the AI component
5.1.1 Isolating the AI

The players in the simulator, in both its old and its new state, move about the playing field according to a basic subsumption architecture. In the original simulator, the Game, Team, and Player classes all worked together to make the players move around the field in such a way that players would avoid running into each other, obey the basic rules of the game, and follow simple offensive and defensive strategic principles. There was no separation between the code that enforced the three different types of actions. In the old simulator, players would simply never choose to run to a place already occupied by another player, nor to throw the disc to land out of bounds. In our simulator, the player may attempt to move and throw in any direction he chooses, and the game physics and rules components adjust players positions if they attempt to occupy the same space and reposition the disc after it lands out of bounds. Collision avoidance will be discussed in the next subsection. Discussion of handling the out of bounds case can be found in [2].

In order to create this layer of separation, the Game, Team and Player classes were designated to handle the physics and rules as shown in Figure 5-1. The Player class contains an API of protected methods to control players actions, plus two abstract methods offense() and defense(), one of which is called every frame depending on whether the player's team is currently on offense or defense. In order to implement the player’s artificial intelligence, the Player class must be subclassed, and the offense() and defense() methods must be overridden. In our project, the offensive and defensive strategies are both implemented in the PlayerE subclass. The player API of protected methods provided to the subclass is discussed in detail in [2].

5.1.2 Collision Prevention

In order to free the artificial intelligence component from the obligation of avoiding collisions, functionality to prevent collisions was added to the physics and rules component. Each frame, after the players move according to the direction they decide to travel in and the place the player physics places them in, the collision detection finds
pairs of players that are less than a certain minimum distance apart, and repositions them. If a moving player runs into a stationary player, the stationary player remains stationary and the moving player alone is repositioned. If two moving players collide, both are moved half the necessary distance to make them sufficiently far apart, as shown in Figure 5-2.

![Diagram](image-url)

Player A is stationary, and player B runs into player A. only player B's position is corrected.  
(a)  

Player A and player B run into each other. Both players' positions are corrected.  
(b)  

Figure 5-2: The effects of collision prevention

5.2 Disc Physics

This version of the simulator offers improved disc physics that cause the frisbee to fly more similarly to how an actual frisbee flies. The Player API includes methods for throwing, catching, and blocking the disc.

5.2.1 Disc Flight

In the old version of the simulator, the disc did not behave realistically. Rather than flying through the air in a way that closely resembles frisbee flight, the disc would travel in a straight line. The disc would originate from a point at chest height at the thrower's position, and travel toward a point on the ground at the thrower's desired
Figure 5-3: The forces that act on a flying frisbee as modeled in this simulation

\[
\begin{align*}
\text{gravity} &= mg \\
\text{lift} &= C_l A \rho v^2 / 2 \\
\text{drag} &= C_d A \rho v^2 / 2 \\
\end{align*}
\]

\[ A = \text{area} \]
\[ C_l = C_{l0} + C_{l0} \alpha \]
\[ C_d = C_{d0} + C_{d0} \alpha^2 \]
\[ \rho = \text{air density} \]

In addition to the forces of gravity, lift, and drag, a real flying frisbee is also subject to torque that causes it to tilt to one side as it travels through the air. However, it was decided that the effects of torque would be ignored. This made the disc travel along a straight line with respect to its x and y coordinates, simplifying the artificial intelligence routine needed to complete passes. The new frisbee physics thereby make throws look like idealized throws that travel through the air without dying off to one side or another. The changes to the disc physics make it possible to throw passes that travel over the heads of other players and then fall down to the height of the receiver. They also make the simulation more visually pleasing, since the disc looks like it flies more like a real frisbee.
<table>
<thead>
<tr>
<th>Throw Type</th>
<th>Power</th>
<th>Height</th>
<th>Angle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>3</td>
<td>0.2</td>
<td>50°</td>
</tr>
<tr>
<td>Normal</td>
<td>5</td>
<td>0.7</td>
<td>0°</td>
</tr>
<tr>
<td>Huck</td>
<td>5</td>
<td>0.7</td>
<td>50°</td>
</tr>
</tbody>
</table>

Table 5.1: The characteristics of different throw types

5.2.2 Throwing

The player API contains a method for throwing called `throwDisc()`. The method calls the private `makeThrow()` method, which executes the actual throw. When `throwDisc()` is called, the player specifies the direction and speed of the disc, as well as whether he wants to throw a normal throw, a low throw, or a huck. As shown in Table 5.1, each throw has a different maximum speed, throwing height, and angle of attack. Player skills are also factored into the throwing process. Throw speed is obtained by multiplying the relative power shown in table 5.1 with the player’s own throwing power rating. The height at which the disc is released is the product of the player’s height and the relative height shown in Table 5.1. Also, the throwing accuracy skill is used to determine whether or not the thrower makes a throwing error. If the player does make a misthrow, the disc’s initial velocity is perturbed by a certain percentage, causing the disc to travel along a slightly different path from where the thrower intended.

The `makeThrow()` method is also called automatically by the `pull()` method, which the physics and rules component calls after each point to cause the possessing player to pull the disc to start each point. When `pull()` is called, the huck type is automatically used, and the disc is aimed in a random direction somewhere between the back two cones of the opposite endzone.

Three protected methods are provided in the Player class to aid the thrower in deciding where to throw. The `getThrowSpeed()` method takes a throw type, a distance, and a time, and returns the speed at which the thrower must throw that type of throw in order for it to reach the specified distance within the specified amount of time. Since the player has a maximum throwing speed, there is a minimum time
which it will take a player to throw a given throw type a given distance, which can be determined by calling the `getMinTime()` method. Likewise, for each throw type and distance, there is a maximum amount of time that the disc can possibly stay in the air and still arrive at the target distance before hitting the ground. This maximum time can be obtained by calling the `getMaxThrowTime()` method of the Player class. By using the `getMaxTime()` and `getMinTime()` methods, a thrower can confirm that throwing a pass a certain distance in a certain time is possible, and then use `getThrowSpeed()` to choose the correct speed at which to throw the disc. The Player API also provides a method called `predictPlayerLocation()` for predicting where a player will be located in a certain amount of time, assuming he continues sprinting in his current direction of travel. The implementation of this method is discussed in [2]. In conjunction, the four methods `predictPlayerLocation()`, `getMinThrowTime()`, `getMaxThrowTime()` and `getThrowSpeed()` can be used to throw a pass that will intersect the path of a moving player.

### 5.2.3 Catching and Blocking

Two methods, `catchDisc()` and `blockDisc()`, are provided for acting on the disc while it is in the air. The `catchDisc()` method allows the player to obtain possession of a flying disc, provided that it is within arms length of his position horizontally, and has a vertical position above the ground and lower than his height plus his arms length. Any player other than the thrower or the defender marking the thrower may attempt to catch the disc. Since it is sometimes possible for the thrower to break the mark and throw the disc to the part of the field obscured by the mark, a different method, `blockDisc()`, is provided for the mark to attempt to make a play on the disc. `catchDisc()` succeeds with probability between 75% and 97.5%, depending on player skill. Similarly, `blockDisc()` probabilistically produces different results. The result will be a point block 20% of the time, causing the disc to stop and fall to the ground. Another 20% of the time, the result will be a deflection, randomly perturbing the flight path of the disc. The remaining 60% of the time, the mark is broken, and the disc’s trajectory is not impacted in any way.
5.3 Defensive Strategy

The defensive artificial intelligence module implements a simple man-to-man defense. Players choose who they will cover differently at different times, depending on the game situation. Defenders also change their positioning relative to the man they are marking based on the offensive player’s position on the field. Defenders occasionally help out their teammates by helping to guard cutters who have beaten their man deep. Lastly, when the disc is in the air, defenders move to intercept it if they calculate that they will be able to reach it before it hits the ground.

5.3.1 Marking Up

Depending on the game situation, defenders will mark their opponents differently. When both teams line up before the pull, defenders will mark up according to height. The tallest defender will guard the tallest opponent, and the shortest defender will guard the shortest opponent. However, when there is a turnover in the middle of the point, players will match up according to position. The player closest to the endzone they are guarding will guard the player that is farthest downfield.

5.3.2 Positioning

The player marking the thrower sets a mark so as to prevent throws to the break side of the field, and the remaining defenders stand near their men, between the man and the disc and slightly toward the open side. All of the downfield defenders stand slightly closer to the disc than their opponent to guard against cuts toward the disc, and the last man back stands behind his defender to guard against deep cuts away from the disc. The mark always forces to the forhand side of the field, which is the right hand side of the field from the point of view of the thrower as he looks downfield. However, when he is far to the force side of the field, he stands more to the middle of the field to prevent cross-field throws, and when he is closer to the break side, he stands more in front of the thrower to prevent throws up the line.
5.3.3 Deep Help

The defender guarding the man closest to the endzone that the offense is trying to score in will stand behind his man in order to get a head start in case his man cuts toward the endzone. In addition, he will also help double team any other offensive player who cuts deep to prevent an easy goal from being scored.

5.3.4 Interceptions

At any time, if a defender calculates that he can make a play at a disc that is in the air, he computes the location at which he can intercept it by calling the computeInterceptLocation() method in the Player class. This is the defender's highest priority, and overrides any other actions he may have been considering. The computeInterceptLocation() method calls the Disc object's predictLocation() method in combination with the Player's predictLocation() method to see if there are any points on the disc's flight path that the player will be able to reach before the disc does.

5.4 Graphical User Interface

Thus far, improvements to the architecture and functionality of the simulator which changed the nature of the game play have been discussed. However, the improvements to the graphical user interface do not add any capability to the simulator, but rather make it easier for the user to understand what is going on in the game.

5.4.1 Automatic Camera

Previously, the software simulation offered four fixed views of the field, plus the ability to rotate, translate and zoom the camera view manually with a 3-button mouse. The user would have to manipulate the camera manually as the game was being played in order to keep up with the action. The new simulator offers three modes of camera viewing. The most basic is the "manual view" previously offered in the old simulator.
The more advanced camera viewing mode, called "fixed view," still requires that the user position the camera manually, but rotates automatically to point toward the disc so that the user can always see the action. Finally, the most advanced camera mode is the "thrower view," which not only automatically points toward the disc, but also positions the camera behind the thrower so that the viewer can see the cuts made by the other players whom the thrower is likely to throw to. A view of the disc as it travels toward a receiver from the point of view of the thrower is shown in Figure ??(a).

5.4.2 Flipped Graphics

One problem with the existing simulator was that the game universe was designed as a left-handed coordinate system. This means that the x, y, and z axes obeyed the left-hand rule rather than the right-hand rule. As a result, the player positions in the 3-dimensional graphics appeared to be inverted along one of the three axes from the positions shown in the overhead view. To fix this problem, a transformation was added to invert all of the graphics along one axis, and the normals of all 3-dimensional objects were generated inward so that the objects would display correctly after being reflected.

5.4.3 Disc Shadow

Certain aspects of the graphics in the old simulator make it difficult to understand how the players and disc are oriented in 3-dimensional space. For example, it is difficult to see how high the disc was off the ground. To compensate, a disc shadow was added to the new simulator so that the viewer can easily determine its height from the 3-dimensional display.

5.4.4 Score Board

Several components were added to the game information display in the score board panel. Previously, only basic game state variables were shown, such as the score,
Figure 5-4: The new user interface display

the possessing team and player, and the stall count. The panel has been modified to show various pieces of information about what the offense and defense are doing. As shown in Figure 5-4(b), arrows show which way the offense is traveling and which way the defense is forcing, and text displays show the most recent actions made by offensive and defensive players. In this figure, the action “MARK BROKEN” shows that player “Player 9” has broken the mark to throw to a receiver on the break side of the field. However, as is indicated by the “MISTHROW” in the event field, he has made a throwing error, and as a result, his throw has not gone exactly where he tried to throw it. The graphical display in Figure 5-4(a) shows the disc landing at the ground a few yards away from the intended receiver’s feet.

Another example of actions and events displayed by the score board from an entirely different run of the simulator is shown in Figure 5-5. The display indicates that player 1 threw the disc to an intended receiver, but that the disc either hit the
ground or landed out of bounds, resulting in a turnover.

5.4.5 User Controls

At the time that the framework of the simulator was modified and the player API was redesigned, the rewind and replay capabilities of the simulator were broken. They have not been fixed. However, functionality to reset the game has been added, which can be enacted by a new button in the simulation control frame. Figure 5-6(a) shows the new simulation controls, which allow the user to reset, play, and pause the game.

In the process of adding automatic cameras that track the disc and the thrower as described in subsection 5.4.1, the camera control frame which allows the user to select the camera mode was also improved. The new camera control frame is shown in figure 5-6(b). The figure shows the possible view locations for the “fixed view” option, where the camera tracks the disc but does not change locations. The “thrower view”
option which automatically repositions the camera behind the thrower, and “manual view” option which allows the user to adjust the camera himself are also shown.
Chapter 6

Lessons Learned

This chapter describes what was learned as a result of doing this project. The focus of this chapter is on critiquing how well the goals and project scope were chosen, and how they might be chosen differently if the project were to be done again.

Although very little of this paper is dedicated to describing the process of modifying the old software simulator to prepare it for the changes that were to be made, this process took between a quarter and a third of our development time. Though we chose our project goals modestly enough to allow time for certain tasks to take longer than planned, the process of changing routine method calls and variable names was definitely the task that was underestimated the most.

Once the framework and basic game play functionality had been implemented and the time came to implement the artificial intelligence components, it was difficult to leave the game play work behind, since there were still so many aspects of the real game of Ultimate Frisbee that had not been incorporated into the simulator. For example, the player with the disc could not pivot, and only had one type of throw rather than a forehand and a backhand, eliminating the phenomenon of the player that can only effectively throw one of the two basic throws. Additionally, since the functionality used to compute the throw speed necessary to throw to a moving player expected that the disc to move in a straight line, curved throws were not possible. Their absence took away the element of curved passes that fly around the defender to the intended receiver. However, despite the disappointment of having a simulator that
was not completely realistic, leaving out these elements of the game greatly simplified the artificial intelligence routines that were needed to play the game. Adding those desired extra physics features might have made the simulator unnecessarily complex and added a lot of work without bringing enough benefit to justify their addition.

One aspect of a real Ultimate Frisbee game that was not supported that turned out to be somewhat problematic was the concept of communication between teammates. When implementing the artificial intelligence routines, the programmer was forced to either come up with algorithms based on the game state for players to execute that would let teammates lead to identical conclusions, or to violate the intended player abstraction and use static variables in the player subclass that all players on the same team could execute. Rather than leaving the programmer with these two undesirable options, the API ought to provide a way for teammates to communicate, similarly to how players might shout to each other during a game.
Chapter 7

Future Work

The simulator presented in this paper has by no means been perfected. There are many opportunities to improve the implementation of the physics, rules of play, artificial intelligence and user interface to make the simulator a more useful learning tool. This section discusses some possible improvements, their usefulness, and how easy or difficult it would be to implement them.

7.1 Framework

Although improvement was made on the original framework, the new simulator's design still has some problems. Although the artificial intelligence routines are mostly separated from the physics and rules component, the enforcement of the physics and the enforcement of the rules are not abstracted from each other at all. While game rules and laws of physics can both be considered to be rules, implementing them in the same module makes it difficult to illustrate the idea of a foul. If a player runs into another player, the simulator ought to be able to indicate that a foul occurred. Implementing fouls would be a difficult task given the current implementation of the simulator.

Even if the physics and rules layers were separated into two software components, there are still some fundamental problems with the division between the artificial intelligence and the rules and physics. Since the artificial intelligence is implemented
by subclassing the Player class, it has access to all public methods in the Player class, including the `update()` method. In theory, this would allow the programmer of the artificial intelligence component to call the Player class’s `update()` method at incorrect times, which should not be allowed. Since the Team class must update the player positions when it is told to, it must be able to call a public method in the Player class, so there is no quick fix to this problem.

7.2 Physics

Although the disc and player physics show a significant improvement over the previous version of the simulator, they are still far from realistic. Players accelerate slowly but can stop instantly, and throws travel in straight lines with respect to their horizontal components. Even if these shortcomings in the implementation of disc and player physics were corrected, players are unable to jump or dive, and there is no concept of wind, pivoting, or forehands and backhands.

There is clearly plenty of opportunity for improving the game’s physics, and the software is written so that such improvements would not be too difficult to make. However, each modification to the physics model requires an increase in the complexity of the artificial intelligence routines needed for the offense and defense to play the game, and careful consideration of the necessity of any new physics features must be made before any are implemented. It must be considered that the task of making a simulator that is useful to an advanced player is much more difficult than making one that is useful to a beginning player, so adding complexity might do more to hurt than to help.

7.3 Artificial Intelligence

While making the physics more complicated would also make the artificial intelligence more complicated, simply implementing a more complex AI routine without making other changes would have no immediately negative side effects. The strategies imple-
mented in this simulator could definitely be improved so that the offense and defense behave realistically more of the time. For example, the thrower could be made to throw deep more often to a player who is wide open in the endzone, and defensive players could be taught to force backhand when the disc is on the far left sideline. Routines could even be added to play zone offense and defense. Due to the abstraction between the physics and rules component and the artificial intelligence component, improving the strategies used by the players is probably the easiest improvement to make to the simulator.

7.4 User Interface

There are many advanced features that could be added to give the user more control over how the game play progressed. The user interface could provide for modification of the defensive and offensive intelligence components to turn on or off the use of certain strategies. This would allow the user to see how the game play varied without certain strategy components, helping to illustrate why those concepts are important. Additionally, more control could be given to the user to modify the teams that are playing in the game. This would allow the user to see how a fast team took advantage of a slow team, how a team with a height advantage took advantage of a shorter team, or how a team’s average throwing ability affected their performance in the simulation. In short, more control could be given to the user to learn about specific game principles, rather than expecting the user to infer how important those strategies are from watching how a match up between a single pair of teams using the same strategies plays out.
Chapter 8

Conclusion

The goal of this simulator is to be a tool that is useful to as many people as possible for learning about the sport of Ultimate Frisbee. While the simulator might not use advanced enough strategies to be useful to an advanced player, it could still be useful to a beginning or intermediate player. Since the simulator does not explicitly explain the basic elements of strategy, a beginning player might only be able to gain an understanding for the rules of the game from using the simulator alone. However, the simulator could be a very useful tool for an advanced player to illustrate basic concepts of strategy to the beginning player. An intermediate player who has a at least a weak understanding of the game’s strategy, however, might be able to learn a fair amount about the game’s strategy from using the simulator on his own. The simulator shows improvement over it’s predecessor, in terms of the flexibility of its design, complexity and accuracy of the game play, as well as the ease of use of the user interface. Future improvements could be made to make the physics more realistic, although these must be made with caution as they will require an increase in the complexity of the artificial intelligence routines. Some improvements could be made to give make it easier for the user to learn about the game’s strategy, such as the ability to turn on and off the use of certain strategies by each team, as well as a more comprehensive display of what each player is doing at a given time. All in all, while there are improvements that can still be made, the project was successful in improving the usefulness of the 3-dimensional software Ultimate Frisbee simulator.
Appendix A

Rules of Ultimate Frisbee

The most recent edition of the rules of Ultimate Frisbee can be found on the worldwide web at: http://www.upa.org/ultimate/rules/rules.shtml. While the actual document detailing the rules of Ultimate Frisbee is quite long, the UPA also provides summary in just 10 rules:

1. The Field – A rectangular shape with endzones at each end. A regulation field is 70 yards by 40 yards, with endzones 25 yards deep.

2. Initiate Play – Each point begins with both teams lining up on the front of their respective endzone line. The defense throws (“pulls”) the disc to the offense. A regulation game has seven players per team.

3. Scoring – Each time the offense completes a pass in the defense’s endzone, the offense scores a point. Play is initiated after each score.

4. Movement of the Disc – The disc may be advanced in any direction by completing a pass to a teammate. Players may not run with the disc. The person with the disc (“thrower”) has ten seconds to throw the disc. The defender guarding the thrower (“marker”) counts out the stall count.

5. Change of possession – When a pass is not completed (e.g. out of bounds, drop, block, interception), the defense immediately takes possession of the disc and becomes the offense.
6. Substitutions – Players not in the game may replace players in the game after a score and during an injury timeout.

7. Non-contact – No physical contact is allowed between players. Picks and screens are also prohibited. A foul occurs when contact is made.

8. Fouls – When a player initiates contact on another player a foul occurs. When a foul disrupts possession, the play resumes as if the possession was retained. If the player committing the foul disagrees with the foul call, the play is redone.

9. Self-Refereeing – Players are responsible for their own foul and line calls. Players resolve their own disputes.

10. Spirit of the Game – Ultimate stresses sportsmanship and fair play. Competitive play is encouraged, but never at the expense of respect between players, adherence to the rules, and the basic joy of play.
Appendix B

Running the Simulator

To run the simulator, the latest Java and Java 3D libraries should be installed. At the time of this writing, Java is at version 1.4 and Java 3D is at version 1.3. Currently, official Java 3D support is provided by Sun for the Microsoft Windows and Sun Solaris platforms. Third-party support is also provided for Linux, SGI IRIX, and HP-UX. Download links are listed below:

- Java library download: http://java.sun.com/j2se/1.4.1/download.html

- Java 3D download (Windows/Solaris):
  http://java.sun.com/products/java-media/3D/download.html

- Java 3D download (Linux):
  http://www.blackdown.org/java-linux/jdk1.2-status/java-3d-status.html

- Java 3D download (IRIX):
  http://www.sgi.com/developers/devtools/languages/java.html

- Java 3D download (HP-UX): http://www.hp.com/products1/unix/operating/

Additional questions on Java 3D may be answered at:
The simulator can then be run by going to

The simulator can also be built from source by anyone with a graphics account at the
Laboratory for Computer Science, or an Athena account at MIT.

The program can be run from a graphics account by executing:

```
    cd /pub/www/ultimate
    ./run
```

Alternately, the latest version of the simulator can be checked out and built by execut-
ing the following commands:

```
    setenv JAVA_OPENGL_NATIVE
    setenv CVSROOT /u2/graphics/projects
cvs checkout ultimate
cd ultimate/ultimate/simulator
make
make run
```

For those with Athena accounts, run the following commands:

```
    setenv CVSROOT ~emerylin/public/cvsroot
cvs checkout ultimate
cd ultimate/simulator
make
make run
```
Appendix C

Source Code

This Appendix contains an excerpt of the PlayerE class, which extends the Player class and implements the player's artificial intelligence. Only the code pertaining to the defensive AI is included. When the player is on defense and it is time for him to act, the rules and physics module calls the `defense()` method, allowing the player to act by calling methods in the Player API. Discussion and source code of the Player API and offensive strategy can be found in [2].

```java
public class PlayerE extends Player {

    // Constructors omitted

    // Offense AI code omitted

    public void defense(double doubleTime) {
        int time = (int)doubleTime;
        state = getGameState();
        myTeam = state.getMyTeamInfo();
        oppTeam = state.getOppTeamInfo();

        // initialize defense if necessary
        if (!defenseInitialized)
            initDefense();

        // compute all necessary state information
        defenseUpdateState();
    }

    57
```
defenseMarkUp();
executeAction(time);
blockOrCatch();

// record variables for next use in next call to defense
lastTimeStamp = state.getElapsedTime();
lastStatus = getGameState().getStatus();
lastDiscIsUp = discIsUp;
lastEndzone = state.getEndzone();
}

// Defense Helper variables
private static final boolean DEBUG = false;
private GameState state = null;
private PlayerInfo[] myTeam = null;
private PlayerInfo[] oppTeam = null;
private double upfieldSide = 0; // unit y direction of this team’s endzone
private double openSide = 0; // unit x direction of the open side
private boolean lastback = false;
private int stackPosition = -1;
private DefensePriority priority = null;
private int markedPlayerId = -1;
private double lastTimeStamp = -10000; // arbitrary large negative number
private GameStatus lastStatus = GameStatus.PRE_PULL;
private boolean defenseInitialized = false;
private boolean defenseForceForehand = true;
private int heightRank = -1;
private int speedRank = -1;
private JLabel[] dLabels = new JLabel[10];
private JLabel label1 = null;
private boolean discIsUp = false;
private boolean lastDiscIsUp = false;
private int lastEndzone = -1;
private double lastMarkTime = -10000;
private int defenseThrowerId = -1;

// Defense Helper Methods
public void resetDefense() {
    lastTimeStamp = -1000; // arbitrary large negative value
    lastStatus = GameStatus.PRE_PULL;


/**
 * Defense helper method that must be called once per point before
 * the defense can function properly.
 */
private void initDefense() {
    if (defenseInitialized) return;

    if (labell == null) {
        labell = new JLabel("");
    }

    heightRank = 0;
    speedRank = 0;
    for (int i=0; i<Constants.NUM_PLAYERS-1; i++) {
        double tmpId = myTeam[i].getId();
        double myId = getId();
        double tmpHeight = myTeam[i].getHeight();
        double myHeight = getHeight();
        double tmpSpeed = myTeam[i].getSkills().getSkill(SkillType.Speed);
        double mySpeed = getSkills().getSkill(SkillType.Speed);
        if (tmpHeight > myHeight || (tmpHeight == myHeight && tmpId > myId)) {
            heightRank++;
        }
        if (tmpSpeed > mySpeed || (tmpSpeed == mySpeed && tmpId > getId())) {
            speedRank++;
        }
    }

    lastEndzone = state.getEndzone();
    defenseInitialized = true;
}

private void defenseUpdateState() {
    discIsUp = true;
    for (int i=0; i<Constants.NUM_PLAYERS; i++)
        if (oppTeam[i].hasDisc())
            discIsUp = false;

    upfieldSide = 1;
    if (state.getEndzone() == Constants.SOUTH)
        upfieldSide = -1;
openSide = upfieldSide;
if (!defenseForceForehand)
  openSide = -1;

lastback = true;
stackPosition = 6;
for (int i=0;i<Constants.NUM_PLAYERS;i++) {
  if (oppTeam[i].getId() == markedPlayerId) {
    for (int j=0;j<Constants.NUM_PLAYERS;j++) {
      if (upfieldSide*oppTeam[j].getPosition().y <= upfieldSide*oppTeam[i].getPosition().y) {
        stackPosition--;
        lastback = false;
      }
    }
  }
  if (oppTeam[i].hasDisc()) {
    defenseThrowerId = oppTeam[i].getId();
  }
}

private boolean isDefenseAtPull() {
  int myEndzone = (state.getEndzone() == Constants.NORTH ? -1 : 1);

  // false if this player is far from goal line
  double ypos = getPosition().y;
  double diff = ypos - myEndzone * Constants.BASEPLAYING_LENGTH/2;
  if (Math.abs(diff) > 1D) return false;
}

  // false if this any teammate is far from goal line
for (int i=0;i<Constants.NUM_PLAYERS-1;i++) {
  double ypos = myTeam[i].getPosition().y;
  double diff = ypos - myEndzone * Constants.BASEPLAYING_LENGTH/2;
  if (Math.abs(diff) > 1D) return false;
}

  // false if any opponent is far from goal line
for (int i=0;i<Constants.NUM_PLAYERS;i++) {
  double ypos = oppTeam[i].getPosition().y;
  double diff = ypos - (-1 * myEndzone) * Constants.BASEPLAYING_LENGTH/2;
  if (Math.abs(diff) > 1D) return false;
private void defenseMarkUp() {
  double time = state.getElapsedTime();
  boolean defenseAtPull = isDefenseAtPull();
  boolean defenseAfterTurnover = (time > lastTimeStamp + 100);

  // don't mark up too often
  if (time < lastMarkTime + 1000) return;

  if (defenseAtPull) {
    defenseMarkUpByHeight();
  } else if (defenseAfterTurnover) {
    defenseMarkUpByPosition();
  }

  lastMarkTime = state.getElapsedTime();
}

private void defenseMarkUpByHeight() {
  if (!defenseInitialized) {
    System.err.println("PlayerE.defenseMarkUp(): defense not initialized");
    System.exit(0);
  }
  Arrays.sort(oppTeam, new HeightComparator());
  markedPlayerId = oppTeam[heightRank].getId();
  System.out.println("Player " + getId() + " marks opponent " + markedPlayerId + ")
}

private void defenseMarkUpByPosition() {
  if (!defenseInitialized) {
    System.err.println("PlayerE.defenseMarkUp(): defense not initialized");
    System.exit(0);
  }

  // compute opponent stack positions
  int[] oppStackPositions = new int[Constants.NUM_PLAYERS];
  for (int i=0; i< Constants.NUM_PLAYERS; i++) {
    for (int j=i+1; j<Constants.NUM_PLAYERS; j++) {

  }
double ipos = oppTeam[i].getPosition().y;
double jpos = oppTeam[j].getPosition().y;
if (-1 * upfieldSide * ipos > -1 * upfieldSide * jpos)
    oppStackPositions[i]++;
else
    oppStackPositions[j]++;
}

// compute this player's relative y position
int relYPos = 0;
for (int i=0;i<Constants.NUMPLAYERS-1;i++) {
    double myYPos = getPosition().y;
    double otherYPos = myTeam[i].getPosition().y;
    if (-1 * upfieldSide * myYPos > -1 * upfieldSide * otherYPos) {
        relYPos++;
    }
}

// assert that above two algorithms worked correctly
for (int i=0;i<Constants.NUMPLAYERS;i++) {
    for (int j=i+1;j<Constants.NUMPLAYERS;j++) {
        if (oppStackPositions[i] == oppStackPositions[j]) {
            throw new IllegalStateException("players " + i + " and " + j + " have same player position");
        }
    }
}

// assign an opponent for this player to mark
for (int i=0;i<Constants.NUMPLAYERS;i++) {
    if (oppStackPositions[i] == relYPos) {
        markedPlayerId = oppTeam[i].getId();
    }
}

private void executeAction(int time) {
    boolean actionChosen = false;

    // consider intercepting the disc

    Point3d discInterceptLocation = null;
    if (discIsUp) {
        long startTime = System.currentTimeMillis();
discInterceptLocation = computeInterceptLocation(time);
}
if (discInterceptLocation != null) {
    setDestination(discInterceptLocation);
    actionChosen = true;
}

// consider helping deep
if (!actionChosen) {
    if (lastback) {

        // identify biggest deep threat
        int deepThreatIndex = 0;
        double downFieldDist = -1 * upfieldSide * oppTeam[0].getPosition().y;
        for (int i=1;i<Constants.NUM_PLAYERS;i++) {
            double dist = -1 * upfieldSide * oppTeam[i].getPosition().y;
            if (dist > downFieldDist) {
                deepThreatIndex = i;
                downFieldDist = dist;
            }
        }
        PlayerInfo deepThreat = oppTeam[deepThreatIndex];

        // consider poaching deep
        if (deepThreat.getId() != markedPlayerId) {
            if (deepThreat.getPosition().y > getPosition().y - 3.0) {
                if (deepThreat.getPosition().distance(getPosition()) > 10) {
                    if (-1 * upfieldSide * deepThreat.getVelocity().y > 0) {
                        markedPlayerId = deepThreat.getId();
                        actionChosen = true;
                    }
                }
            }
        }
    }
}

// otherwise, mark
if (!actionChosen) {
    defenseChaseMark();
    priority = DefensePriority.MARK;
}
private class HeightComparator implements Comparator {
    public int compare(Object a, Object b) {
        int result = Double.compare(((PlayerInfo)b).getHeight(),
            ((PlayerInfo)a).getHeight());
        if (result == 0)
            result = ((PlayerInfo)b).getId() - ((PlayerInfo)a).getId();
        return result;
    }
    public boolean equals(Object o) {
        return o instanceof HeightComparator;
    }
}

private void defenseChaseMark() {
    if (!defenseInitialized) {
        System.err.println("defense not initialized in PlayerE.defenseChaseMark()");
        System.exit(0);
    }
    PlayerInfo[] oppTeam = state.getOppTeamInfo();
    for (int i=0;i<Constants.NUMPLAYERS;i++) {
        if (oppTeam[i].getId() == markedPlayerId) {
            Point3d destination = oppTeam[i].getPosition();
            if (oppTeam[i].hasDisc()) {
                double ratio = 0.5 - openSide * destination.x/Constants.BASE_FIELD_WIDTH;
                double angle = -3*Math.PI/4 + ratio * Math.PI/4;
                double discSpace = 2.5;
                destination.add(new Point3d(discSpace*openSide * Math.cos(angle),
                    discSpace*upfieldSide * Math.sin(angle),
                    0));
            } else {
                boolean lastback = true;
                for (int j=0;j<Constants.NUMPLAYERS;j++)
                    if (-1*upfieldSide*oppTeam[j].getPosition().y
                        > -1*upfieldSide*oppTeam[i].getPosition().y)
                        lastback = false;
                if (lastback) {
                    destination.add(new Point3d(1.5*openSide, -1.5*upfieldSide, 0));
                } else {
                    destination.add(new Point3d(1.5*openSide, 1.5*upfieldSide, 0));
                }
            }
        }
    }
}
private Point3d computeInterceptLocation(double inc) {
    double armLength = getHeight() * Constants.ARM_LENGTH_RATIO;
    double hReach = armLength;
    double vReach = getHeight() + armLength;
    for (double time = inc; time < 10000; time += inc) {
        Point3d target = predictDiscLocation(time);
        if (target.z <= 0D) break;
        double timeToCatch = 1000 * timeToPoint(target);
        if (timeToCatch < time && target.z < vReach) {
            target.set(target.x, target.y, 0);
            return target;
        }
    }
    // interception not possible
    return null;
}

private void blockOrCatch() {
    if (defenseThrowerId == markedPlayerId) {
        blockDisc();
    } else {
        catchDisc();
    }
}

Bibliography


