Developing an Intelligent Ultimate Frisbee Player

by

Emery Lin

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

May 21, 2003

[June 2003]

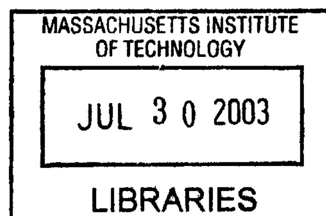
Copyright 2003 Emery Lin. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author ____
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by ________
Seth Teller
Thesis Supervisor

Accepted by ________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 3 0 2003
LIBRARIES

BARKER

Developing An Intelligent Ultimate Frisbee Player
by
Emery Lin

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The goal of this thesis project was to develop an intelligent agent capable of playing ultimate frisbee, a popular disc sport. An execution component capable of handling the physical and technical aspects of the game, such as running, throwing, and game rules, was first developed. Using the application programming interface (API) to this component, an intelligent agent capable of playing the game was developed. Finally, a graphical interface featuring two- and three-dimensional representations of the game was made. The ultimate simulator was developed in conjunction with David Bailey. This thesis project discusses the player physics module in the execution component, the API to the execution component, and the development of the offensive portion of the intelligent agent, while Bailey's thesis discusses the frisbee physics module, the development of the defensive portion of the intelligent agent, and the design of the graphical interface.

Thesis Supervisor: Seth Teller
Title: Associate Professor Of Computer Science & Engineering

Acknowledgements

I'd like to thank my advisor, Seth Teller, for his support and guiding hand in this project, and David Bailey for working on this project with me. Without their insight this project could not have been completed.

I'd also like to thank my family and friends for supporting me throughout my five years at MIT.

Table of Contents

1 Introduction 6
1.1 The Game of Ultimate 6
1.2 Previous Work 7
1.3 Thesis Overview 9
2 Design Goals 10
2.1 Player Attributes 10
2.2 Application Programming Interface 12
2.3 Offensive Player Module 12
3 Implementation 14
3.1 Language and Software Platform 14
3.2 Player Attributes 14
3.3 API Implementation 19
3.3.1 Player Movement 19
3.3.2 Game State 21
3.3.3 Java Implementation 23
3.4 Offense Implementation 28
3.4.1 Throwers 30
3.4.2 Receivers 33
4 Results 37
5 Future Work 40
Appendix A: Running The Simulator 42
Appendix B: Selected Source Code 43
Bibliography 67

Table of Figures

Figure 1-1: field of play 6
Figure 3-1: player speed chart 15
Figure 3-2: player acceleration model 16
Figure 3-3: player cutting model 17
Figure 3-4: player reaction time chart 18
Figure 3-5: Player class 24
Figure 3-6: the GameState object 27
Figure 3-7: sample team definition file 28
Figure 3-8: handle rating computation 29
Figure 3-9: illustration of thrower terms 30
Figure 3-10: open factor illustration 32
Figure 3-11: illustration of receiver terms 35
Figure 4-1: throwing to a poached receiver 37
Figure 4-2: throwing to an open receiver 38

1 Introduction

1.1 The Game of Ultimate

Ultimate Frisbee is a fast-moving, non-contact disc sport played by two teams of seven players. The basic premise of the game is to score points by passing the disc into an endzone. Typical dimensions of the field are shown in figure 1-1.

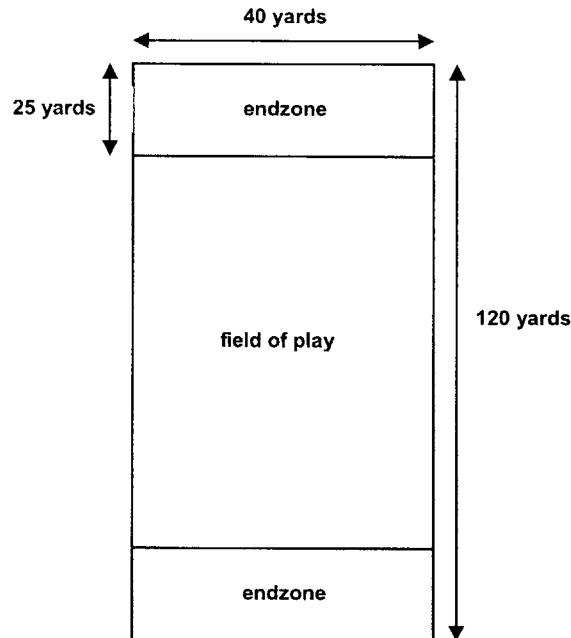


Figure 1-1: field of play

To start the game, a player from one team *pulls*, or throws the disc as far as possible, down the field to opposing team. Once the opposing team receives the disc by catching it or picking it up from the ground, they attempt to advance down the field by passing the disc from player to player.

A player in possession of the disc is not allowed to move, and is given ten seconds to pass in any direction to a teammate. A team commits a *turnover* when a thrown disc fails to be caught within the field of play, or when the disc is not thrown within the ten second limit. The opposing team then gains possession, and the opportunity to score.

The most important rule in ultimate frisbee is the spirit rule, which requires players to value fair, clean, and honest play.

Like other sports, ultimate is played both casually and seriously. The game is easy to learn, but difficult to play at a high level. At the top level of the sport, players must execute on offense and defense with precision.

1.2 Previous Work

In their infancy, computer games featured weak computer opponents that were easy to defeat. Modern computer opponents are now capable of producing sophisticated, dynamic behavior and providing a reasonable challenge to the human player. Artificial intelligence labs have spent a great deal of time developing computer agents capable of playing popular games like *Quake*, *Descent*, and *NBA Live!* at a high level.

No advanced computer agent currently exists for the sport of Ultimate Frisbee. Ultimate enjoys a wide following throughout the world, and the level of play at the top level is breathtaking to watch. However, no professional Ultimate leagues exist, and commercial attenion to the game is accordingly minimal.

Dean Bolton started work on a ultimate simulator for his Master's thesis in the fall

of 2001. The system he produced focuses more on developing a useful graphical interface than on producing realistic Ultimate players. As such, the game play in Bolton's simulator is too simplistic and needs to be improved.

Much work has been done in developing intelligent computer agents for sports games like *NBA Live* and *Madden*, and first-person shooter games like *Quake* and *Descent*. Some of these efforts are geared toward developing challenging computer opponents for human players, while other efforts are focused on developing agents capable of playing one-player games.

While there has been minimal work done for Ultimate in particular. Rodney Brooks' work on subsumption architectures could prove useful. In a subsumption architecture, varying levels of control are built into an agent; each successive level of control is more advanced than the last. An agent must satisfy the requirements of the lowest control layer before being able to consider the requirements of the layer above. Bolton used the idea of a subsumption architecture, but did not fully realize its potential in developing an intelligent player.

The Soar/Games project at the University of Michigan has focused on making computer games more fun to play by creating intelligent computer agents. Creating these agents often provides insight into other topics in artificial intelligence such as machine learning and interface design. The Soar Quakebot, which plays the computer game *Quake*, follows a decision cycle called the perceive, think, act loop. In the perceive stage, the agent accepts sensor information from the game. In the think stage, the agent processes this information, and decides what to do based on the knowledge it has. In the act stage, the agent executes external and internal actions. This type of decision cycle

could prove useful in the development of an Ultimate player.

1.3 Thesis Overview

Three components were developed for the ultimate simulator: an execution layer and an application programming interface (API) to use it, a decision layer utilizing the API to create an intelligent player, and a graphic interface featuring both two-dimensional and three-dimensional representations of the game. The simulator was developed in conjunction with David Bailey, also a Master's student at MIT.

This thesis consists of five chapters. Chapter 2 presents the design goals of the simulator. Chapter 3 discusses the implementation of the simulator. The results of the simulator are presented in Chapter 4, and Chapter 5 discusses areas for future work. This thesis focuses on the player execution layer, the API for the execution layer, and the implementation of the offensive part of the ultimate player. Other aspects of the simulator are discussed briefly, but explained in full detail in Bailey's thesis, which covers the frisbee execution layer, the implementation of the defensive part of the ultimate player, and the graphic interface used for the simulator.

2 Design Goals

2.1 Player Attributes

Ultimate players are not all equal. In order to promote interesting gameplay and realism, it is necessary to be able to define varying levels of skill. The following physical skills are important in defining an ultimate player:

- **Speed:** the easiest skill to recognize, but not necessarily the most important. Smart players lacking above-average speed can still be extremely effective.
- **Acceleration:** players with good acceleration have an easier time gaining separation from their opponents
- **Reaction time:** players with good reaction time can more easily defend opponents, as well as respond to what is happening on the field
- **Cutting ability:** being able to cut effectively is one of the most important skills an ultimate player can possess. A slow player who gets open consistently is far more useful than a fast player who doesn't know how to utilize his speed. A good cutter is able to change directions quickly while retaining most of his speed.

In addition to physical skills, a player must be able to work with the frisbee effectively. There are two traditional ways to throw a frisbee.

The most popular and easiest way to throw is with the backhand motion. The player wraps four fingers around the bottom of the disc, places his thumb on top, and then uncoils his arm across his body. The complement to the backhand is the forehand motion,

which most people find more difficult to execute. The player grips the disc by placing his middle finger along the rim of the disc, index finger perpendicular to the middle finger pointing toward the center of the disc, and thumb above the disc, and releases it by flicking his wrist forward. Because most players throw backhands more effectively, most defenders will try to force the thrower to throw forehand by standing in such a manner that makes it difficult to make the backhand motion.

The following frisbee skills are important in defining an ultimate player:

- **Accuracy:** most players can throw to a still target, but in ultimate the targets are rarely standing still. Accurate throwers are able to hit players in stride, minimizing the chance of an interception by the defender. Players generally have different levels of accuracy with their forehand and backhand.
- **Power:** accuracy generally decreases with the distance a frisbee is thrown. Being able to throw long distances is important. When pulling to start a point, the farther the frisbee goes, the farther the opposing team must travel in order to score. On offense, the ability to throw long stretches the defense and reduces the number of throws it takes to score. A typical player is able to throw farther backhand than forehand.
- **Catching ability:** catching a frisbee is usually an afterthought, but catches in ultimate games are typically made at full speed, under tremendous pressure from a defender. Sure-handed catchers breed more confidence in throwers.

2.2 Application Programming Interface

A good application programming interface (API) is necessary in the design of the ultimate simulator. The purpose of the API is to provide access to an execution layer that handles the physical and technical aspects of the game. The execution layer handles tasks such as running, throwing, cutting, and catching based on a player's attributes. This layer also ensures that the game maintains a valid state, and exposes this state to the decision-making module.

It is important to separate execution from decision because a player's decisions do not always coincide with his actions. Throws do not always travel to their intended target or travel far enough, and players cannot always run fast enough to catch the disc. It is also important to have a common execution layer to prevent one decision module from being superior to another due solely to the physics implementation.

The goal of the API is to be clean and easy to use. It should provide all of the tools necessary to program an effective team of players.

2.3 Offensive Player Module

The goal of the offensive portion of the decision module is to use the API to produce a player capable of playing a realistic ultimate frisbee game. Ultimate is a unique game in that every player on offense can be called upon to play any role. Unlike football, there are no purely offensive or defensive players. Unlike basketball, players do not have firmly set positions. While players may be assigned nominal positions at the start of a

point, the continuous and flowing nature of the game ensures that more often than that, players are unable to maintain these positions. Unlike soccer, players are not assigned to specific portions of the field. There are limitless situations in which a player may find himself.

One approach to the problem is to write code for a number of specific situations in a game, and try to account for as many situations as possible. However, this approach is tedious and hard to maintain. Another approach is to try and write a small set of universal rules that players follow at all times. However, this approach may produce an unrealistic looking game. The qualities of ultimate make it a difficult game to program.

3 Implementation

3.1 Language and Software Platform

The simulator has been programmed with the Java and Java3D libraries. Java and Java3D are available on a wide variety of platforms, including Windows, Solaris/Sparc, Linux, and SGI IRIX. The simulator has been written as an applet, so that it can be viewed through a web browser as well as a standalone application.

3.2 Player Attributes

A total of nine player attributes are used in defining a player: forehand accuracy, backhand accuracy, forehand power, backhand power, catching ability, speed, cutting ability, reaction time, and acceleration. Proficiency in an attribute is defined on an integer scale of zero to ten, with ten being the best and zero the worst. The physical attributes are discussed here. The main goal of simulator is not to provide a perfect model of the physical world. Therefore, the goal of the physics implementation is to be realistic without being overly complicated.

The speed attribute determines the top speed that a player can achieve. The scale is defined linearly so that a player with a speed rating of one has a top speed of 6 yards/second, and one with a speed rating of ten has a top speed of 8 yards/second. The scale is illustrated in Figure 3-1. A world class sprinter can maintain a speed of approximately 11 yards/second over ten seconds. Therefore, the defined scale is reasonable for a typical set of athletes.

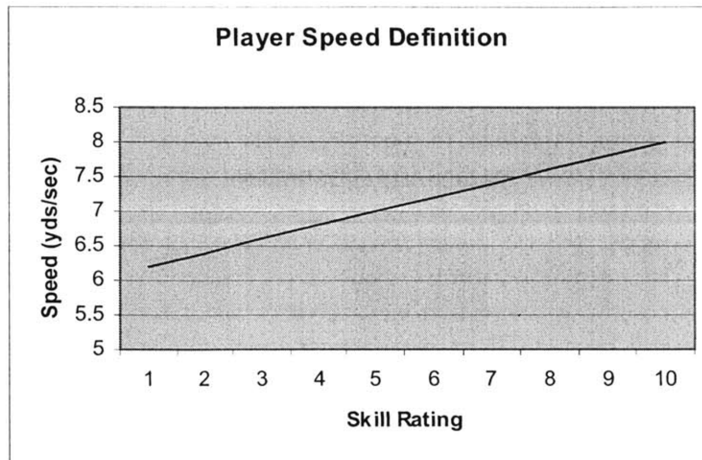


Figure 3-1: Player speed chart

The acceleration phase of a player is divided into two stages: burst and after-burst. This model reflects the fact that a player can start running very quickly, but takes time to reach his top speed. In the burst stage, a player is able to reach a certain percentage of his top speed in a short amount of time. In the after-burst stage, a player is able to accelerate to his full speed.

The acceleration attribute determines how long it takes a player to reach his top speed. A player with an acceleration rating of zero takes five seconds to reach his top speed, while a player with a rating of ten takes four seconds to reach his top speed. The scale increments linearly.

The burst stage consumes the first 500 milliseconds of the acceleration stage. During this time, a player accelerates linearly and is able to reach 75% of his top speed. The after-burst stage consumes the remainder of the acceleration period, which is variable with a player's acceleration rating. During this time, the player also accelerates linearly to his top speed. After the acceleration period is over, a player is able to maintain

his top speed indefinitely. Again, this approach was deemed to be a sufficient compromise between realism and simplicity. The acceleration model is illustrated in figure 3-2.

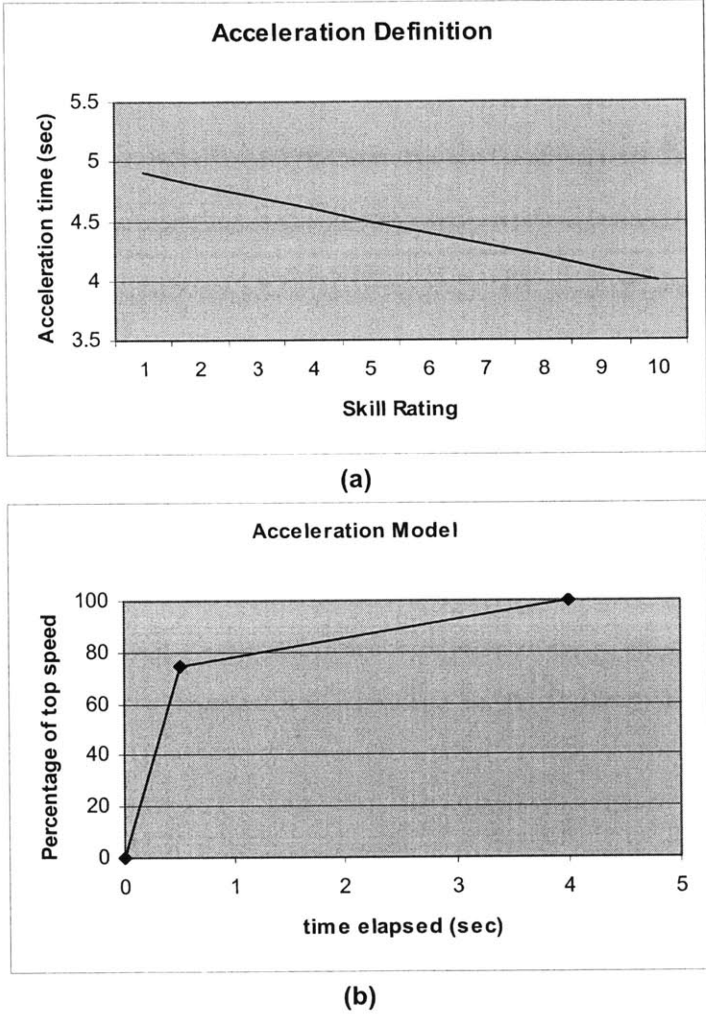
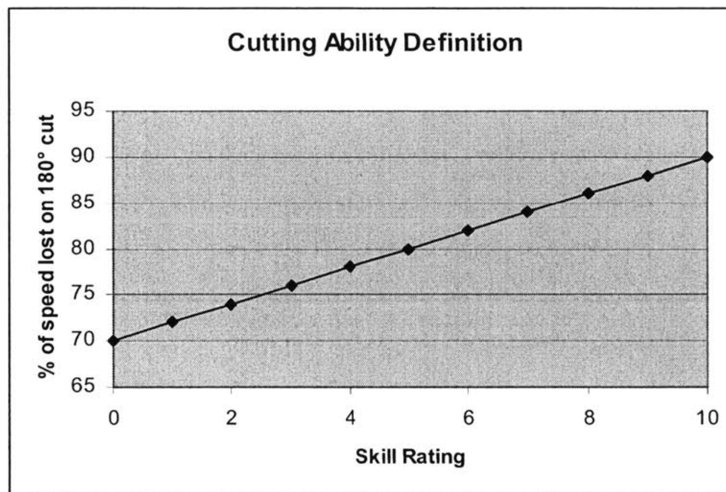


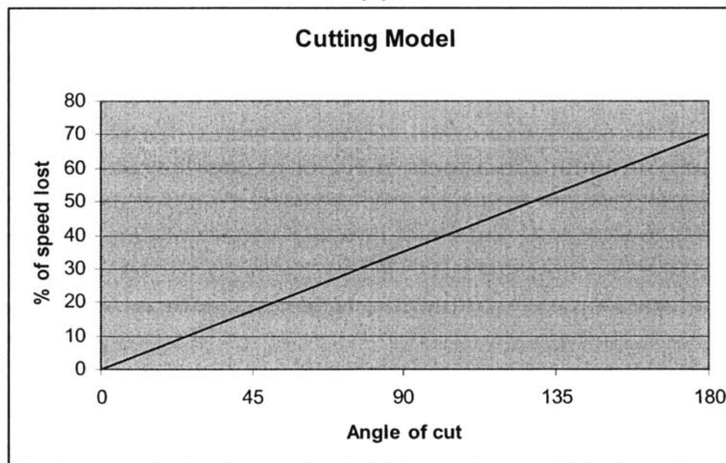
Figure 3-2: Player acceleration models: (a) acceleration time by rating; (b) percentage of top speed attained for a player with an acceleration rating of 10

Being able to cut effectively plays a large role in being an effective offensive ultimate player. Players who make good cuts are able to change direction quickly and gain separation from defenders. The cutting ability attribute determines how much speed

a player is able to retain when he changes direction. A player with a rating of zero loses 90% of his speed when he reverses direction, i.e. makes a 180° change in direction, while a player with a rating of ten loses 70% of his speed when he reverses direction. The percentage of speed lost increases linearly with the angle of the cut, from 0° to 180°. The cutting model is illustrated in Figure 3-3.



(a)



(b)

Figure 3-3: Playing cutting model: (a) percentage of speed lost on a 180° cut by rating;(b) percentage of speed lost for a playing with cutting ability of 10

The final physical attribute represented in the simulator is reaction time. Players with faster reaction time start their own actions and respond more quickly to what they see on the field. A player with a rating of zero has a reaction time of 700 milliseconds. Reaction time decreases linearly, and a player with a rating of ten has a reaction time of 300 milliseconds. The reaction time scale is illustrated in Figure 3-4.

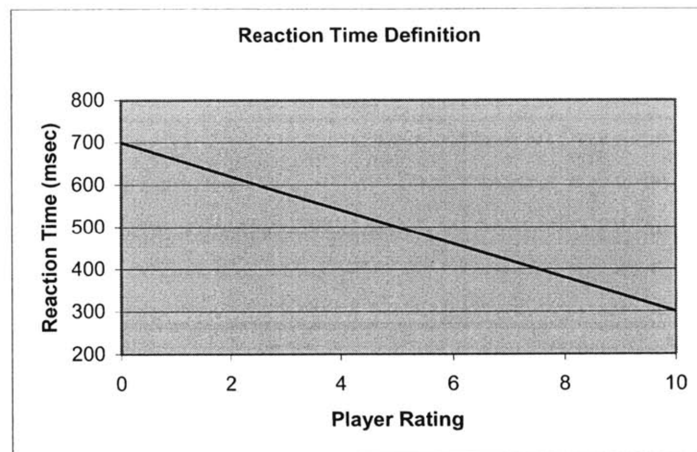


Figure 3-4: player reaction time chart

Reaction time is necessary to ensure that players execute actions in a reasonable manner. For example, while an offensive player knows where he is going, a defensive player must constantly monitor the offensive player to decide where to go. It is therefore not realistic to have a defender be able to follow his target without an appropriate response time. To illustrate how reaction time works, suppose an offensive player A and a defensive player B both have a reaction time of 500 milliseconds. Suppose player A decides to move down the field at timestamp 0. Because of his reaction time, he will not start his movement until timestamp 500. Player B detects A's movement at timestamp 500, and decides to follow him. Because of B's reaction time, he will not be able to start following A until timestamp 1000. A therefore gains a half second advantage on B.

3.3 API Implementation

As outlined previously, the purpose of the API is to provide an execution layer that handles the physical and technical aspects of an ultimate frisbee game. Physical aspects include running, throwing, catching, and cutting. Technical aspects include keeping track of the score, the stall count, possession, and making sure the game remains in a valid state. This section discusses how running, cutting, and rules of the game are handled by the execution layer, and then how they are implemented in Java.

3.3.1 Player Movement

The execution layer is responsible for making sure that players move in a valid manner during gameplay. This means making sure a player does not move instantaneously from one point to another, run and cut beyond his ability, move while in possession of the disc, or react too quickly to an event.

In order to prevent impossible movement, the API does not provide methods to set the position or velocity of a player. Instead, the API provides access to a player's travel path. The decision module specifies a desired destination by adding a location or series of locations to the travel path. At any time, a player can either append a location to his current travel path, set a new destination, or remove all destinations from his travel path.

When a location is added to an empty travel path, the execution layer waits for the player's reaction time to elapse before moving the player toward the location. When a

location is appended to a non-empty travel path, the execution layer does not wait for the reaction time to elapse, since the player has pre-determined a path of travel.

When the travel path is populated, the execution layer moves the player toward the first location in the path according to his current velocity and position. If the player is not moving, he enters the burst stage of acceleration, and accelerates until he either reaches his top speed or reaches his destination. Once the destination has been reached, the execution layer checks whether there is another location in the travel path. If the path is empty, the player stops at the destination until another location is added. If another location exists in the path, the execution layer computes how much speed the player will lose based on the angle of his cut.

Following the cut, the player is either in the middle of the burst stage or after-burst stage. If the player is in the burst stage, the execution layer determines how much time is left in the burst stage by computing the ratio of his current speed to his top burst speed. If the player is in the after-burst stage, then he accelerates at his after-burst level to his speed. This method of movement ensures that all players follow the defined acceleration model at all times.

The execution layer removes all locations from the travel path if the player gains possession of the disc, and does not allow locations to be added to the travel path while a player is in possession of the disc.

3.3.2 Game State

The execution layer is also responsible for making sure that the game is always in a valid state, and providing the game's state to any decision-making module that requests it.

When the game starts, the players must be lined up in the correct position on the endzone before the pull is executed. During this time, the execution layer sets the travel paths of all players to positions on the endzone so that the game may start. The pull is then executed, and normal game play begins.

Once game play has started, the decision layer takes over control of player movements. The decision layer can then request of the state of the game in order to determine how to best move.

The API makes the following data available to the decision layer: the stall count, game status, the current endzone, disc information, player information, and elapsed time in the game.

The stall count is fundamental to the game. An offensive player might use the stall count to determine when to throw the disc, and a defensive player might use it to determine how to position himself. The stall count is given in milliseconds, and ranges from 0 to 10000 by default.

The game status lets a player know what is currently going on in the game. Examples of game status include before the pull, during the pull, normal gameplay, and goal scored. The current endzone, either north or south, tells the player which endzone he

is currently defending. The disc information received by the player consists of the position, velocity, and normal vector of the disc.

The player receives information about two sets of players: his teammates and his opponents. The information included in these sets are slightly different. Both sets of information include position, velocity, height, a unique id, and a flag indicating whether a player is in possession of the disc. However, a player's skill level is only disclosed if he is a teammate. The reasoning behind this decision is that a player most likely knows his teammates' skill level, but is not likely to know his opponents'. All opponents appear as having average ratings in all categories.

Finally, the player is provided with the elapsed time of the game. A player could use this information to track how recently a specific event occurred.

The API provides several additional methods to assist the player in making decisions. These methods allow the player to predict the location of another player at some future time, predict the location of the disc at some future time, calculate the length of time needed to run to a location, calculate minimum and maximum times possible for throwing to a location, and calculate speed needed to throw to a location within a specific time. The API provides these methods so that a decision layer's effectiveness is based solely on game decisions.

The execution layer is responsible for tracking when a turnover occurs during game play. A turnover occurs when no one catches a thrown disc, an opposing player catches a thrown disc, a teammate catches a thrown disc out of bounds, or when the stall count has expired. When a turnover occurs, the execution layer changes the possession of the disc to the opposing team.

Sometimes the decision layer must temporarily cede control of player movements in order for the game to remain in a valid state. For example, when an offensive player catches the disc in his opponent's endzone, a point is scored and the game must reset. When this happens, the execution layer moves the players to valid positions on the endzone so that the next point can start.

The execution layer must also take control during certain types of turnovers. Play cannot start in the endzone, except directly following a pull. Therefore, if a player picks up a disc in either endzone, or catches a disc in his own endzone, the disc must be walked back into the field of play before the game can resume.

When these types of turnovers occur, the execution layer populates the travel path with the appropriate location and allows the player with the disc to move. The decision layer is temporarily suspended until travel is completed.

3.3.3 Java Implementation

The core of the execution layer is implemented in four main classes: `Game`, `Team`, `Player`, and `Disc`.

The `Player` class contains the bulk of execution layer. It controls physical aspects of the player, and provides methods for moving, throwing, and catching, as well as the methods of prediction outlined previously. A player must have both an execution layer and a decision layer, but only the execution layer is provided by the API. Therefore, the `Player` class is implemented as an abstract class. It is the responsibility of an

implementing subclass to define the decision layer. The relevant variables and methods of the `Player` class are shown in Figure 3-5.

```
abstract class Player {

    variables
    Vector travel               Vector3d velocity
    Point3d position            double height
    PlayerSkill skills          int id

    Abstract methods
    offense
    defense

    Execution layer methods
    action
    decide
    update

    API methods
    setDestination              addDestination              getDestination
    clearTravelPath

    catchDisc                   throwDisc                   pickupDisc
    blockDisc                   getMinThrowTime             getMaxThrowTime

    predictLocation             timeToPoint                 getThrowSpeed
    predictDiscLocation         getGameState

}
```

Figure 3-5: Player class

The Player class contains variables which store the physical properties of the player, such as velocity, position, height, and skills. Each player also has a unique id so that they can be identified consistently across multiple refreshes. Methods are also provided for interfacing with the disc.

The travel path discussed previously is implemented as a `Vector` of `TravelPoints`. A `TravelPoint` contains a `destination`, a `timestamp`, and two boolean flags called `used` and `remove`. The `timestamp` indicates when the `TravelPoint` is eligible to be used. The `used` flag indicates whether the player has traveled toward the

destination, and the `remove` flag indicates whether the player wishes to discard the `TravelPoint`.

When a new destination is set, `timestamp` is set to the current game timestamp, plus the player's reaction time. When a new destination is added to a travel path that already contains one or more destinations, the timestamp is set to the current game timestamp. The `used` flag is set to true as soon as the player starts traveling toward the destination.

The purpose of the `remove` flag is not immediately obvious. Since the refresh rate is typically much faster than a player's reaction time, it is not always possible to simply remove a `TravelPoint` from the travel path as soon as it is not needed. Suppose, for example, that the refresh rate is 50 milliseconds, and a player's reaction time is 500 milliseconds. It is possible for a player to set a new destination at every refresh. Suppose a player sets new destinations at timestamps 0, 50, 100, and 150. The valid timestamps for this destinations are then 500, 550, 600, and 650, respectively. However, if the player clears his travel path at timestamp 300, then none of the four destinations will have been used. Therefore, instead of removing `TravelPoints` immediately, the `remove` flag is set to true. A `TravelPoint` is only eligible to be removed from the travel path when both the `remove` and `used` flags are true.

Four methods are used to interface with the travel path: `setDestination`, `addDestination`, `clearTravelPath`, and `getDestination`. The `setDestination` method invalidates all previous `TravelPoints`, while `addDestination` appends to the current list of `TravelPoints`. The `getDestination` returns the first location in the travel path if it exists and its timestamp is valid.

Every time the simulator is refreshed, the `action` method in Player is called. The `action` method first calls the `decide` method, which in turn calls either `offense` or `defense`, which are both abstract methods left for a subclass to implement. The `offense` and `defense` methods implement the decision layer for a player. Once `decide` has been executed, the `update` method updates the player's position according to his travel path. If `getDestination` returns a location, then the player proceeds to move to that point using the movement model outlined previously.

The `Team` class stores a set of `Player` objects in an array, and stores information common to all members of a team, such as which endzone the team is currently defending, if the team currently has possession of the disc, and if so, which player currently possesses the disc.

The `Game` class is responsible for controlling game play. It contains two instances of the `Team` class, called `redTeam` and `blueTeam`. When the game is running, the `Game` class ensures that the game maintains a valid state, and stores information such as the score, the stall count, and current possessing team.

`Game` calls the appropriate methods in `Team` and `Player` depending on the state of the game. Before the pull and after a goal has been scored, `Game` will line up all the players on the endzone so that the game can start. During game play, `Game` handles special situations described previously, such as when the disc lands in the endzone, or out of bounds. When a turnover occurs, `Game` changes the disc possession so that the appropriate methods in `Player` get called.

The `Disc` class contains method for controlling the flight of the disc, and is discussed in detail in Bailey's thesis.

As mentioned previously, the API makes the following information available to the decision layer: the stall count, game status, the current endzone, disc information, player information, and elapsed time in the game. This information is stored in a class called GameState (Figure 3-6). The decision layer receives a GameState object when the getGameState method is called.

The `GameStatus` class is an enumeration class containing static instances for simple identification of the game status, such as `GameStatus.PULL`. Note that rather than returning `Disc` and `Player` classes for the disc and player information, the `Trajectory` and `PlayerInfo` classes are used. `Trajectory` and `PlayerInfo` are wrapper class implementations of `Disc` and `Player`, respectively. The purpose of these classes is to restrict the information available to the decision layer. For example, a player is not allowed to know the skill level of an opposing player, but `Player` contains a public method for getting a player's skills.

```
class GameState {

      double stallCount
      double elapsedTime
      GameStatus status
      int endzone
      Trajectory discInfo
      PlayerInfo[] teammates
      PlayerInfo[] opponents

}
```

Figure 3-6: The GameState object

The API supports the definition of player skills through text files. This enables the same decision code to be executed with players with varying levels of skill in different areas of the game. The file format allows for the definition of up to seven players and their skill level.

Each player is defined on a line starting with the text `player`. Eleven integers must then follow, indicating the player's height (in inches), skill level in forehand power, forehand accuracy, backhand power, backhand accuracy, catching ability, speed, cutting ability, reaction time, stamina, and acceleration. Stamina is currently not used by the execution layer, but is included for possible future development.

Failure to provide eleven valid integers following any line beginning with `player` invokes an `IllegalFileException`. When this exception occurs, the game automatically generates two teams of players who are average in every skill. A sample team definition file is shown in Figure 3-7.

```
; comments may be inserted here

player 60 6 5 5 5 5 5 5 5 5 5
player 60 7 6 8 9 4 5 3 1 4 4
player 60 5 5 5 5 5 5 5 5 5 5
player 60 5 5 5 5 5 5 5 5 5 5
player 60 5 5 5 5 5 5 5 5 5 5
player 60 5 5 5 5 5 5 5 5 5 5
player 60 9 9 9 9 5 5 5 5 5 5
```

Figure 3-7: sample team definition file

3.4 Offense Implementation

The goal of the decision module presented here is to create an intelligent player capable of playing a varied game with a minimal set of rules. While it might be tempting to write a piece of code to account for every conceivable situation in a game, this approach is both tedious and difficult to maintain.

The approach presented here utilizes the most popular offensive strategy in ultimate, which is the stack offense. The underlying principle behind the stack is that it is difficult to complete passes when the field is crowded with players. With fourteen players

running in a relatively small space, the defense quickly gains the advantage. The goal of the stack offense is to reduce the amount of congestion on the field by lining up the players in a straight line called the stack. The stack structure opens up the areas to the left and right of the thrower. Players in the stack can then work to get open in these areas. If they are unsuccessful in receiving the disc, the retreat back in the stack to allow another player to cut.

In a well run stack, the stack moves fluidly down the field with the disc, the field is free of congestion, and the thrower has a constant flow of open cutters trying to receive the disc.

The decision module keeps track of the location of the disc in a variable called `pivot`. When the game starts off of a pull, the offensive players predict where the disc will be in ten seconds, and `pivot` is assigned to this location. The player with the best overall disc handle rating travels to the pivot, while the rest of the team lines up in the stack. The handler rating is determined by the formula outlined in Figure 3-8. The rating weights accuracy more than power, and rewards players who are proficient in both the backhand and forehand throw.

```
fa = forehand accuracy
fp = forehand power
ba = backhand accuracy
bp = backhand power

handleSkill = (fa + 0.5fp) * (ba + 0.5bp)
```

Figure 3-8: handle rating computation

The best handler always lets the disc drop to the ground before picking up the disc, to avoid the possibility of dropping the disc. Dropping the disc while receiving the

pull results in a turnover and advantageous field position for the opposition. Once the player has the disc, he begins to look for teammates to throw to.

3.4.1 Throwers

In general, throwers look for moving targets to throw to. Players who stand still are usually not open, because a nearby defender can easily run to and intercept the disc. Moving players are more attractive targets because they are able to run and maintain separation with their defender. The decision-making module uses this principle in its throwing decisions.

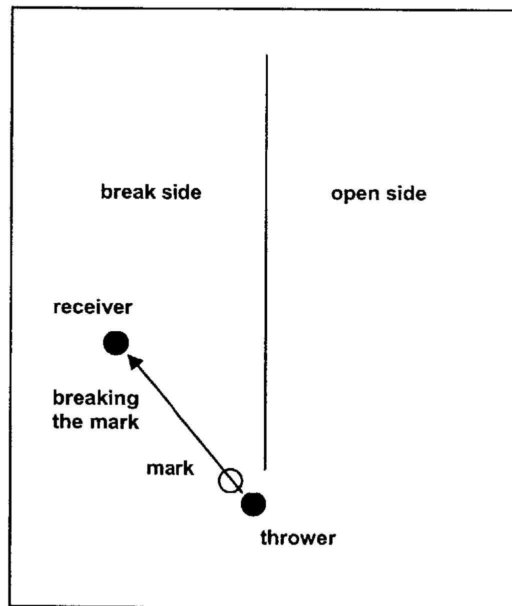


Figure 3-9: illustration of thrower terms

The defender guarding the disc, called the *mark*, will usually try to make the thrower pass to one half of the field, commonly called the *open side*. The defender concedes throws going toward the open side, and concentrates his efforts on preventing

throws from going to the opposite half of the field, called the *break side*. A good marker allows the defense to concentrate on guarding potential receivers on the open side, instead of the entire field. While it is tempting to try and prevent any throw at all, this is nearly impossible in practice, and the effectiveness of the defense is minimized when it has to guard receivers across the entire field. Trying to throw to the break side around the defender is called *breaking the mark*. Breaking the mark is advantageous for the offense, because defenders will usually be caught on the open side, leaving plenty of space on the break side for receivers to catch the disc. Illustrations of these terms are shown in figure 3-9.

With this type of defense in mind, the thrower uses the following metric to determine how open a receiver is. For a given location amount of time in the future, the thrower analyzes the predicted position of the receiver, and the predicted position of each opposing player by using the `predictLocation` API call. The thrower checks to see the distance between each opposing player and the line formed by him and the receiver, and the distance between each opposing player and the receiver. The minimum of these fourteen distances (two for each opposing player) is designated the *open factor* for a receiver. The computation of open factor and an illustration of open and non-open receivers is provided in figure 3-10.

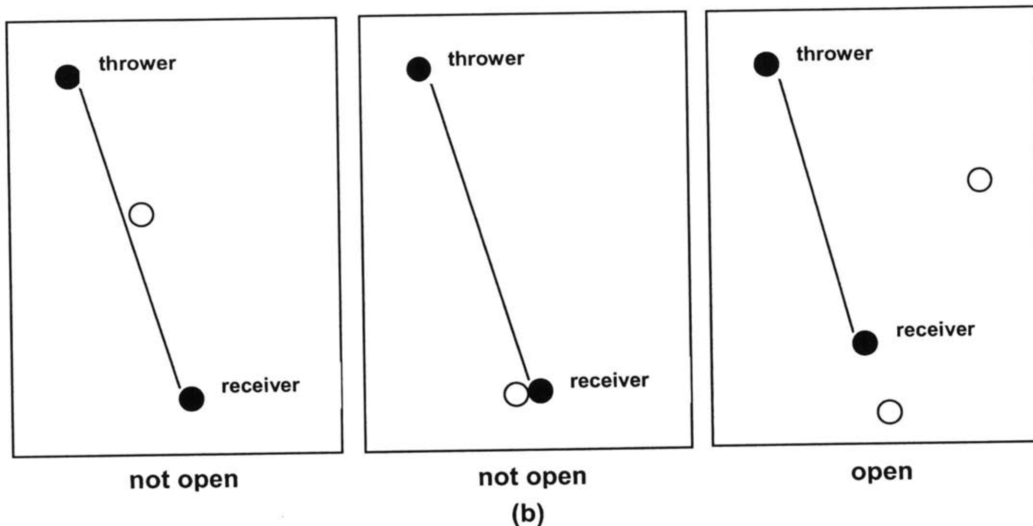
One shortcoming of this method is that receivers on the break side will almost never appear to be open because the mark is usually very close to the line between thrower and receiver. This is of course what the mark is designed to do. However, this approach ignores receivers who are otherwise extremely open. To address this issue, the thrower ignores the mark twenty percent of the time when computing the open factor for

a receiver. This allows the thrower to take some chances on the break side. The disc will sometimes be deflected or blocked when the thrower tries to throw by the mark.

```
T = thrower
R = receiver
L = line which goes through T.position and R.position
O = open factor

for each member X of opposing team {
        d1 = distance between X.position and R.position
        d2 = distance from X.position to L
        O = min(O, d1, d2)
}
```

(a)



(b)

Figure 3-10: open factor: (a) computation (b) illustration

While moving players are usually better targets for throwers, sometimes a non-moving target will be extremely open, or *poached*. In these situations, the thrower should hit the non-moving target. The decision module defines a player who has an open factor of 5 or greater to be poached.

The decision process for the thrower scans non-moving receivers first. For each receiver in this group, the thrower will compute the minimum time it would take to throw

to him, using the `getMinThrowTime` API call, and compute the open factor for the player at this future time. If any non-moving receiver is poached, the thrower will throw to him, breaking ties by throwing to the receiver with the highest open factor.

If no one is poached, the thrower proceeds to analyze the moving receivers. For each potential receiver, the thrower predicts his location 500 to 2500 milliseconds in the future, stepping by 50 milliseconds, and checks to see whether a throw can made to the target in the given amount of time. If the throw is feasible, the thrower then computes the open factor for that future time.

The thrower repeats this process for each moving teammate, and stores the location and time corresponding to the maximum open factor. While this process will always return a receiver as long as one player is moving, the receiver will not necessarily be open. Since there are ten full seconds to throw, the thrower will not throw the disc with more four seconds left on the stall count if the maximum open factor does not exceed 1.

3.4.2 Receivers

The priority for receivers in this decision module is to minimize congestion, or clogging, on the field, and provide good cuts for the thrower. Since a good cut is useless if another player is cutting at the same time, minimizing clogging is the highest priority for a receiver.

As mentioned previously, the `pivot` variable keeps track of the location of the disc whenever it is caught, and is reset only when the disc is caught in a new location.

Whenever `pivot` is set, a vector called `bisect` is also established. Like its name implies, `bisect` splits the area of the field in front of the thrower in half.

When the pull is received, receivers use `pivot` and `bisect` to determine where to stack. The first receiver in the stack lines up 10 yards downfield from the thrower on bisect. Each subsequent receiver lines up 5 yards behind the receiver in front of him on the bisect vector. The order of the stack is based on each receiver's id, which is arbitrarily chosen at the time the player is initialized. By lining up on the `bisect` vector, the receivers give themselves equal space to work on both sides of the stack.

Several definitions are necessary before receiver movements are explained. From the thrower's perspective, the area between the right sideline and the stack line is defined to be the *right side* of the field, and the area between the left sideline and the stack line is defined to be the *left side* of the field. A receiver standing on the left side of the field is *clogged left*, and a receiver standing on the right side of the field is *clogged right*.

A player is said to be *cutting left* if he is on the left side of the field, and his current travel path would intersect the left sideline if extended indefinitely in a straight line. A player is said to be *cutting right* if he is on the right side of the field, and his current travel path would intersect the right sideline if extended indefinitely in a straight line. The left side of the field is clogged when any receiver is clogged left or cutting left, and similarly for the right side of the field. A player is *farthest back* if he is the farthest player standing vertically from the disc. These definitions are illustrated in figure 3-11.

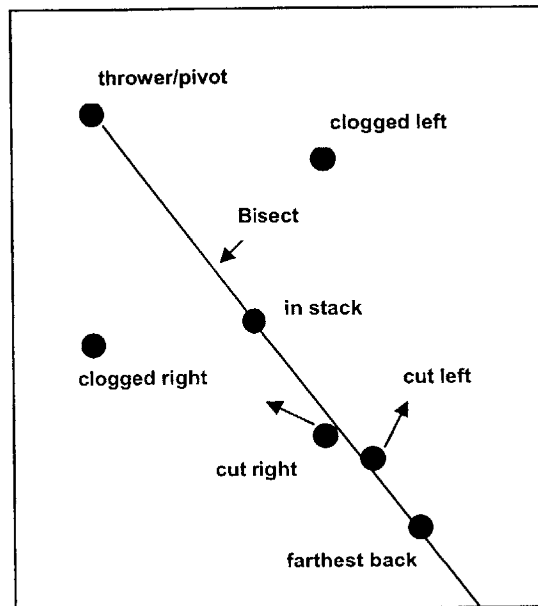


Figure 3-11: illustration of receiver terms

At each game refresh, a receiver determines if he is the farthest back. If he is, he then checks whether the left and right sides of the field are clogged. If exactly one side of the field is clogged, the receiver proceeds to cut to the opposite side. If neither side of the field is clogged, the receiver randomly picks a side to cut to. If both sides of the field are clogged, the receiver does nothing.

Each receiver subsequently checks to see if he is either clogged left or clogged right, regardless of whether or not he is farthest back. If either condition is true, then the receiver must head back to the stack line. A player will finishes a cut and does not receive the disc automatically becomes clogged.

Since all receiver movements are predicated on the pivot, when the pivot changes, all receivers' travel paths are cleared. At this point, most if not all receivers are clogged. On the next game refresh, the receivers will head to the new stack line, with the exception

of the receiver who is farthest back, who proceeds to make his next cut. The process then restarts and continues until a goal is scored or the disc is turned over.

A receiver has two options when he is designated to make a cut. He can either *cut in*, or toward the thrower, or *cut deep*, away from the thrower. Deep cuts gain more yardage when completed, but long passes are harder to complete. In cuts have a better chance of being completed, but if the defense is aware that no deep passes are being thrown, they may start to put all their players closer to the thrower to cut off all in cuts. To keep the defense aware of long throws, receivers cut deep approximately 15% of the time, and cut in the rest of the time.

The receiver movements are designed to minimize clogging on the field while maximizing opportunities for receivers, by making sure only one receiver is cutting into each side at a time, and everyone else is either standing in the stack line waiting to cut, or making a movement to become unclogged.

4 Results

Since the API to the execution layer and the decision layer were developed by the same programmers, the API was constantly changed and developed. As a result, if either the offensive or defensive module needed a missing piece of information, this information was added to the API. For example, the elapsed timestamp of the game was not initially provided. However, when Bailey determined that his defenders might want to know how long ago they were on offense, the timestamp was added to the `GameState` object.

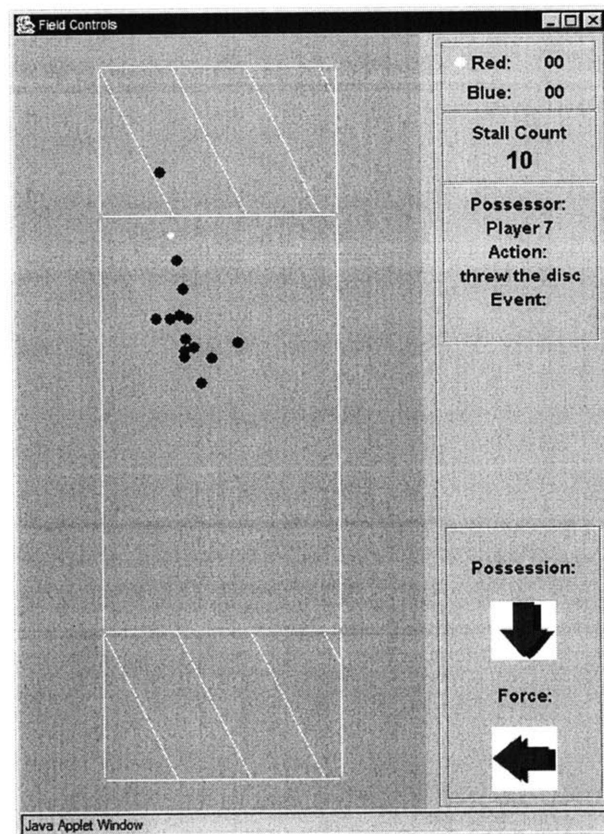


Figure 4-1: throwing to a poached receiver

The game play results were encouraging. Realistic, competitive games emerged from the rules governing the offense and defense. Players on offense were able to

complete series of passes and score with regularity, but also suffered lapses in scoring due to passes that were misthrown, deflected, or blocked, or passes thrown to receivers who were not truly open, as happens in a real game. Throwers try to break the mark, and occasionally try to throw long passes to score quickly. Figure 4-1 above shows a player throwing to a non-moving receiver who is poached. This normally occurs right after a pull, since defenders are rarely able to run fast enough down the field to cover the thrower and the first player in the stack. Figure 4-2 shows a player throwing to a receiving cutting in, about to be open.

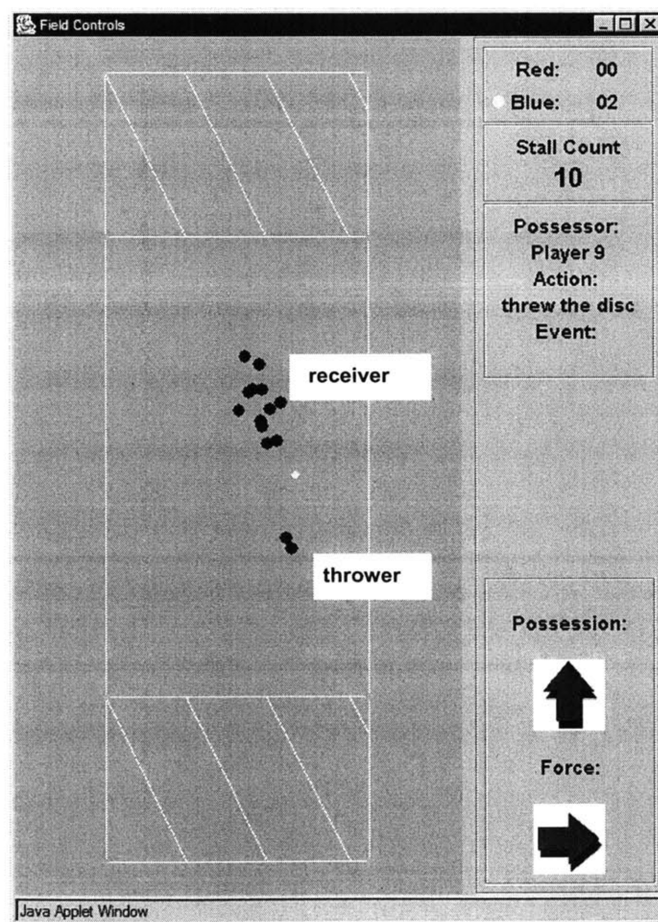


Figure 4-2: throwing to an open receiver

The game play in the simulator approximately matches the level of a set of average players, who are versed in the structure of ultimate. However, the simulator is not able to emulate the precise and structured offense and defense that the world's top teams are able to use.

The level of play is indicative of just how much a player is processing when he is on the field. For example, the open factor that the thrower uses in the simulation uses hundreds of computations to decide on an appropriate receiver. Yet, these computations might only be a fraction of what an experienced thrower actually uses while on the field. This observation is fundamental to why it remains nearly impossible to create an intelligent agent capable of challenging an experienced human player despite the abundance of computing power available.

5 Future Work

There are many possible directions in which to take the simulator. The current implementation of offense and defense is just one of many philosophies that teams currently use.

For example, the zone defense is quite common in the game. In a zone defense, instead of guarding a specific man, defenders guard a specific area. This defense negates the stack offense, since defenders are already clogging the entire field. In response, the offense must also convert to a zone offense. In the most common variation of the zone offense, players called handlers move the disc from one side of the field to another, trying to find a weakness in the zone to cutting players downfield. The purpose of a zone defense is to force the offense to complete a large number of passes in order to score.

Another type of offense is called the split-stack offense. The split-stack uses the same philosophy as the stack of not clogging the field. However, instead of trying to create space by lining up the stack in the middle of the field, players create space by lining up on the side of the field, leaving the middle of the field open.

The split-stack is more difficult to play and execute than the regular stack, and is usually used by more experienced teams. The most common way for defenders to react to the split-stack is to clog the middle of the field to try and prevent throws.

Many different offensive and defensive modules could be written for the simulator.

Another way to extend the simulator would be to allow human game play, turning it into more of a traditional game than just a viewable interface. This extension would present its own set of challenges, such as graphics and ease of play issues.

Hopefully, growing interest in the game of ultimate will spur more interest in developing the simulator.

Appendix A: Running The Simulator

To run the simulator, the latest Java and Java3D libraries should be installed. At the time of this writing, Java is on version 1.4 and Java 3D is on version 1.3. At this time, official Java3D 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 Java3D may be answered at http://java.sun.com/products/java-media/3D/java3dfaq-1.3.html.

The simulator can then be run by going to http://graphics.lcs.mit.edu/~emerylin/UltimateApplet.html.

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.

For those with graphics accounts, run the following commands:

```
setenv CVSROOT /u2/graphics/projects
setenv JAVA_OPENGL_NATIVE
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 B: Selected Source Code

Selected source from the simulator is included here for perusal. The full source may be obtained from a CVS repository, outlined in Appendix A.

The following code is from the `Player` class, which forms the core of the execution layer.

```
package ultimate.simulator;

import javax.media.j3d.*;
import javax.vecmath.*;
import java.util.*;

/**
 * <p>The Player class represents an Ultimate player.</p>
 *
 * <p>The Player class is declared abstract because while it
 * provides functions for the physical components of a Player,
 * it is left to a subclass to implement the decision-making
 * process.
 */

public abstract class Player {

  private Random random = new Random(System.currentTimeMillis());

  /**
   * The player's position on the field.
   */
  private Point3d position;

  /**
   * A list of points denoting the player's future destinations
   */
  private Vector travel;

  // tracker booleans
  private boolean specialState;
  private static boolean firstPickup;

  // attributes
  private boolean hasDisc;
  private double direction;
  private double height;
  private Hashtable flightDataTable;
  private Hashtable throwPowerTable;
  private int id;
  private int reactionTime;
  private int teamId;
  private Player3D player3D;
  private PlayerSkill skillSet;
  private String name;
  private Team team;
  private Vector3d velocity;

  private double lastBlockTime;

  // private physical variables
  private double topSpeed;
  private double accelTime;
  private double burstAcc;
  private double burstSpeed;
  private double afterAcc;
```

```
private double maxPctLoss;

// amount of time spent on current running path
private double time;
private GameStatus gameStatus;

// Constructors

/**
 * Initialize a player with the specified id, skills, height, and
 * position with zero velocity and no travel points.
 * @param id this player's id number
 * @param skillSet this player's skills
 * @param height the height of this player
 * @param position the position of the player on the field
 */
public Player(int id, PlayerSkill skillSet, double height,
  Point3d position) {
  height /= 36; // convert from inches to yards
  this.id = id;
  this.skillSet = skillSet;
  this.height = height;
  this.position = position;
  this.velocity = new Vector3d();
  this.travel = new Vector();
  specialState = false;
  firstPickup = true;
  time = 0;
  setPhysicalConstants();
  initFlightData();
}

private Player(Player player) {
  this.id = player.id;
  this.teamId = player.teamId;
  this.position = new Point3d(player.position);
  this.travel = (Vector)player.travel.clone();
  this.skillSet = new PlayerSkill(player.skillSet);
  this.velocity = new Vector3d(player.velocity);
  this.time = player.time;
  this.team = player.team;
  this.specialState = player.specialState;
  setPhysicalConstants();
}

public void reset() {
  travel.clear();
  specialState = false;
  firstPickup = true;
  velocity = Constants.ZERO_VELOCITY;
  time = 0;
  resetOffense();
  resetDefense();
  updateGraphics();
}

public abstract void resetOffense();

public abstract void resetDefense();

// Public Accessor Methods

/**
 * Get the angle of the direction that the player is facing measured
 * in radians from standard position.
 */
public double getDirection() {
  return direction;
}

/**
```

```
 * Get the default direction for this player to face.
 */
public double getDefaultDirection() {
  return (team.getEndzone() == Constants.NORTH ? Math.PI/2 : -Math.PI/2);
}

/**
 * Get the height of the player.
 * @return the height of the player in inches.
 */
public double getHeight() {
  return height;
}

/**
 * Get the id of the player.
 */
public int getId() {
  return id;
}

/**
 * Get the team id of the player.
 */
public int getTeamId() {
  return team.getId();
}

/**
 * Get the endzone of the player.
 */
public int getEndzone() {
  return team.getEndzone();
}

/**
 * Get the name of the player.
 */
public String getName() {
  return name;
}

/**
 * Sees if the player has the disc.
 * @return true if the player possesses the disc, false otherwise
 */
public boolean hasDisc() {
  return hasDisc;
}

public void setHasDisc(boolean hasDisc) {
  this.hasDisc = hasDisc;
}

/**
 * Get the position of this player.
 * @return a Point3d representing the coordinates of the player on
 * the field.
 */
public final Point3d getPosition() {
  return new Point3d(position);
}

/**
 * Get the velocity of this player.
 * @return a Vector3d representing the x-,y-, and z-velocity of the
 * player.
 */
public final Vector3d getVelocity() {
  return new Vector3d(velocity);
}
```

```
/**
 * Get the skill set of this player.
 * @return a PlayerSkill object representing this player's skills.
 */
public final PlayerSkill getSkills() {
  return new PlayerSkill(skillSet);
}

/**
 * Sets the team of this player.  This method must be called in
 * order to complete the initialization of the player's state, and
 * must only be called once.
 */
public void init(Team team, Point3d position, double direction) {
  if (this.team != null) {
    System.err.println("Error: Team already set in Player.init()");
    System.exit(0);
  }
  this.team = team;
  this.teamId = team.getId();
  this.name = "Player " + getId();
  this.position = position;
  this.direction = direction;
  initGraphics(position, direction, height, teamId);
}

private void initGraphics(Point3d position, double direction, double
                          height, int teamId) {
  Point3d graphicsPosition = new Point3d(position);
  graphicsPosition.scale(Constants.GRAPHICS_3D_MULTIPLIER);
  direction -= Math.PI/2;
  player3D = new Player3D(graphicsPosition, direction, height, teamId);
}

/**
 * Line up the player in the specified position on the endzone.
 * Only valid when the game is in PRE_PULL state.
 */
public void goToPullPosition(int order) {
  firstPickup = true;
  if (team.getGame().getStatus() == GameStatus.PRE_PULL) {
    travel.clear();
    if (team.getEndzone() == Constants.NORTH) {
      setDestination(Constants.NORTH_PULL_POS[order - 1]);
    } else {
      setDestination(Constants.SOUTH_PULL_POS[order - 1]);
    }
  }
}

/**
 * Set the player in the specified position on the endzone.
 */
public void setAtPullPosition(int order) {
  travel.clear();
  if (team.getEndzone() == Constants.NORTH) {
    position = Constants.NORTH_PULL_POS[order - 1];
  } else {
    position = Constants.SOUTH_PULL_POS[order - 1];
  }
}

/**
 * Line up the player on the specified order on the endzone.
 * Only valid when Game is in PRE_PULL state, otherwise does nothing.
 */
public void lineUp(int order) {
  if (team.getGame().getStatus() == GameStatus.PRE_PULL) {
    if (team.getEndzone() == Constants.NORTH) {
      position = Constants.NORTH_PULL_POS[order - 1];
```

```
      } else {
        position = Constants.SOUTH_PULL_POS[order - 1];
      }
    }
    travel.clear();
    velocity = Constants.ZERO_VELOCITY;
    time = 0;
  }

  // only allow opposing players to see an average-skill player
  private PlayerInfo oppPlayerInfo() {
    return new PlayerInfo(id, position, velocity, height, new
                          PlayerSkill(), hasDisc);
  }

  private PlayerInfo samePlayerInfo() {
    return new PlayerInfo(id, position, velocity, height, new
                          PlayerSkill(skillSet), hasDisc);
  }

  private PlayerInfo getPlayerInfo(Player player) {
    if (player.team == this.team) {
      return new PlayerInfo(player, false);
    } else {
      return new PlayerInfo(player, true);
    }
  }

  public TransformGroup getTransformGroup() {
    return player3D.getTransformGroup();
  }

  public String toString() {
    return position.toString();
  }

  // Protected and Private Methods

  public final void setDestination(Point3d dest) {
    TravelPoint tp = new TravelPoint(dest, team.getGame().getElapsedTime() +
Constants.REACTION_TIME);
    if (travel.size() != 0) {
      for(int x=0; x<travel.size(); x++) {
        TravelPoint temp = (TravelPoint)travel.get(x);
        temp.setRemove();
      }
    }
    travel.add(tp);
  }

  // for endzone pickups
  private void setSpecialDestination(Point3d dest) {
    travel.clear();
    TravelPoint tp = new TravelPoint(dest, team.getGame().getElapsedTime());
    travel.add(tp);
  }

  public final void addDestination(Point3d dest) {
    if (travel.size() == 0) {
      setDestination(dest);
    } else {
      TravelPoint tp = new TravelPoint(dest, team.getGame().getElapsedTime());
      travel.add(tp);
    }
  }

  public Point3d getDestination() {
    if (travel == null || travel.size() == 0) {
      return null;
    } else {
      TravelPoint point = (TravelPoint)travel.get(0);
```

```
      if (point.isRemoved() && point.isUsed()) {
        travel.remove(0);
        return getDestination();
      } else {
        if (point.getTimeStamp() <= team.getGame().getElapsedTime()) {
          point.setUsed();
          return point.getPoint();
        } else {
          return null;
        }
      }
    }
  }

  public final void clearTravelPath() {
    travel.clear();
    time = 0;
  }

  protected void pull() {
    if (hasDisc) {
      Vector2d vector2d = null;
      double speed = 45;
      if (team.getEndzone() == Constants.NORTH) {
        vector2d = getRandomPull(speed, Constants.SECorner, Constants.SWCorner);
      }
      else {
        vector2d = getRandomPull(speed, Constants.NWCorner, Constants.NECorner);
      }
      makeThrow(ThrowType.HUCK, vector2d);
      team.getGame().setAction("pulled the disc", teamId);
      hasDisc = false;
    }
  }

  /**
   * @return a random pull at the given speed in a direction chosen
   * randomly betwen the direction from this player to p1 and the
   * direction from this player to p2.  the direction will be chosen
   * from the arc starting at p1 and following in a clockwise
   * direction toward p2.
   */
  private Vector2d getRandomPull(double speed, Point3d p1, Point3d p2) {
    double a1 = PointMath.getXYAngle(position, p1);
    double a2 = PointMath.getXYAngle(position, p2);
    if (a1 > a2) {
      System.err.println("Warning: angles fixed in Player.getRandomPull");
      a2 += 2*Math.PI;
    }
    double r = random.nextDouble();
    double angle = r * (a2 - a1) + a1;
    return new Vector2d(speed * Math.cos(angle), speed * Math.sin(angle));
  }

  /**
   * Have the player attempt to pickup the disc.
   * @return true if successful, false otherwise
   */
  protected boolean pickup() {
    Disc disc = team.getGame().getDisc();
    if (disc.getPosition().z == 0D && disc.getVelocity().length() == 0D) {
      double armLength = height*Constants.ARM_LENGTH_RATIO;
      int possessingTeamId = team.getGame().getPossessingTeam().getId();
      if (position.epsilonEquals(disc.getPosition(), armLength)  &&
          (teamId == possessingTeamId || (teamId != possessingTeamId &&
           team.getGame().getStatus() == GameStatus.PRE_PULL))) {
        System.out.println(getId() + " picked it up. firstpickup is " + firstPickup);
        disc.setStationaryPosition(new Point3d(position.x, position.y,
height*Constants.DISC_HEIGHT_RATIO));
        team.getGame().setPossessingPlayer(this);
        team.getGame().setThrower(this);
```

```
        hasDisc = true;
        if (team.getGame().getStatus() == GameStatus.PLAY &&
            Constants.inEndzone(position) && !firstPickup) {
          System.out.println("in special state");
          setSpecialDestination(Constants.closestEndzonePosition(position));
          specialState = true;
        } else {
          if (team.getGame().getStatus() == GameStatus.PLAY) {
            team.getGame().startStallCount();
          }
        }
        if (!(team.getGame().getStatus() == GameStatus.PRE_PULL)) {
          firstPickup = false;
        }
        return true;
      }
    }
    return false;
  }

  /**
   * Return if any player on team possesses disc.
   */
  protected boolean someoneHasDisc() {
    if (team.getGame().getPossessingPlayer() == null) return false;
    else return true;
  }

  /**
   * Have the player attempt to catch the disc.
   * @return true if successful, false otherwise
   */
  protected final boolean catchDisc() {
    if (this == team.getGame().getThrower() || someoneHasDisc()) {
      return false;
    }
    Disc disc = team.getGame().getDisc();
    Point2d position2d = new Point2d(position.x, position.y);
    Point3d discPosition = disc.getPosition();
    Point2d discPosition2d = new Point2d(discPosition.x, discPosition.y);
    double armLength = height*Constants.ARM_LENGTH_RATIO;
    if (discPosition2d.epsilonEquals(position2d, armLength)) {
      double discHeight = discPosition.z;
      if (discHeight > 0 && discHeight < armLength + height) {
        boolean interception = (teamId != team.getGame().getPossessingTeam().getId());
        team.getGame().setPossessingPlayer(this);
        team.getGame().setThrower(this);
        if (interception) {
          team.getGame().setEvent("INTERCEPTION");
          if (team.getGame().getStatus() == GameStatus.PLAY &&
              ((team.getEndzone() == Constants.NORTH &&
Constants.inNorthEndzone(position)) ||
               (team.getEndzone() == Constants.SOUTH &&
Constants.inSouthEndzone(position))) &&
              !firstPickup) {
            System.out.println("in special state");
            setSpecialDestination(Constants.closestEndzonePosition(position));
            specialState = true;
          }
          System.out.println("interception");
        }
        team.getGame().setAction("caught the disc");
        updateStationaryDiscPosition();

        time = -Constants.REACTION_TIME;
        if (!specialState) {
          clearTravelPath();
          team.getGame().startStallCount();
        }
        return true;
      }
```

```
    }
    return false;
  }

  protected final void blockDisc() {
    if (someoneHasDisc()) {
      return;
    }
    Disc disc = team.getGame().getDisc();
    Point2d position2d = new Point2d(position.x, position.y);
    Point3d discPosition = disc.getPosition();
    Point2d discPosition2d = new Point2d(discPosition.x, discPosition.y);
    double armLength = height*Constants.ARM_LENGTH_RATIO;
    if (discPosition2d.epsilonEquals(position2d, armLength)) {
      double discHeight = discPosition.z;
      if (discHeight > 0 && discHeight < armLength + height &&
          team.getGame().getElapsedTime() > lastBlockTime + 500) {
        double blockThreshold = 0.2;
        double deflectThreshold = 0.4;
        double r = random.nextDouble();
        if (r < blockThreshold) {
          team.getGame().setAction("POINT BLOCK", teamId);
          Trajectory t = disc.getTrajectory();
          Point3d p = t.getPosition();
          Vector3d v = t.getVelocity();
          Vector3d n = t.getNormal();
          v.set(0, 0, -10);
          disc.setTrajectory(new Trajectory(p, v, n));
        }
        else if (r < deflectThreshold) {
          team.getGame().setAction("DEFLECTION", teamId);
          Trajectory t = disc.getTrajectory();
          Point3d p = t.getPosition();
          Vector3d v = t.getVelocity();
          Vector3d n = t.getNormal();
          deflectThrow(v, 0.25);
          disc.setTrajectory(new Trajectory(p, v, n));
        }
        else {
          team.getGame().setAction("MARK BROKEN");
        }
        lastBlockTime = team.getGame().getElapsedTime();
      }
    }
  }

  private final void updateStationaryDiscPosition() {
    double x = position.x;
    double y = position.y;
    double z = height*Constants.DISC_HEIGHT_RATIO;
    y += Constants.DISC_SPACE_RATIO * Math.sin(direction);
    x += Constants.DISC_SPACE_RATIO * Math.cos(direction);
    team.getGame().getDisc().setStationaryPosition(new Point3d(x, y, z));
  }

  protected final void throwDisc(ThrowType throwType, Vector2d vector2d) {
    if (hasDisc()) {
      // this block controls the frequency and magnitude of throwing errors.
      int throwSkill = skillSet.getSkill(SkillType.ForehandAcc);
      if (random.nextDouble() > 0.45 + 0.05 * throwSkill) {
        team.getGame().setEvent("MISTHROW");
        perturbThrow(vector2d, 0.15);
      }
      team.getGame().setAction("threw the disc");
      makeThrow(throwType, vector2d);
    }
  }

  private void makeThrow(ThrowType throwType, Vector2d vector2d) {
    Disc disc = team.getGame().getDisc();
    Point3d discPosition = disc.getPosition();
```

```
    discPosition.z = height * throwType.getHeight();
    disc.setTrajectory(new Trajectory(discPosition, throwType.getAngle(), vector2d));
    System.out.println(getId() + " threw the disc.");
    hasDisc = false;
    team.getGame().setPossessingPlayer(null);
    team.getGame().stopStallCount();
  }

  private void perturbThrow(Vector2d v, double ratio) {
    double angle = random.nextDouble() * 2 * Math.PI;
    double mag = ratio * v.length();
    System.out.print("Throw perturbed: old throw = " + v);
    v.set(v.x + mag * Math.cos(angle),
          v.y + mag * Math.sin(angle));
    System.out.println(" new throw = " + v);
  }

  private void deflectThrow(Vector3d v, double ratio) {
    double angle = random.nextDouble() * 2 * Math.PI; // h. angle of deflection
    double drop = random.nextDouble(); // vertical drop of deflection
    double mag = ratio * v.length(); // magnitude of deflection
    System.out.print("throw deflected: old v = " + v);
    v.set(v.x + mag * Math.cos(angle),
          v.y + mag * Math.sin(angle),
          v.z - drop * mag);
    v.scale(1/(1 + ratio)); // ensures that deflection does not speed up disc
    System.out.println(" new v = " + v);
  }

  // for when the player makes a turn
  public final void dropDisc() {
    System.out.println(getId() + " dropped the disc.");
    hasDisc = false;
    team.getGame().setPossessingPlayer(null);
    team.getGame().stopStallCount();
    team.getGame().getDisc().setStationaryPosition(position);
  }

  protected final GameState getGameState() {
    PlayerInfo myTeamPlayers[] = new PlayerInfo[6];
    for(int x=0;x<Constants.NUM_PLAYERS;x++) {
      Player player = team.getPlayer(x);
      if (!(this == player)) {
        insertSorted(myTeamPlayers, player);
      }
    }
    PlayerInfo oppTeamPlayers[] = new PlayerInfo[7];
    Team oppTeam;
    if (team == team.getGame().getRedTeam()) {
      oppTeam = team.getGame().getBlueTeam();
    } else {
      oppTeam = team.getGame().getRedTeam();
    }
    for(int x=0;x<Constants.NUM_PLAYERS;x++) {
      Player player = oppTeam.getPlayer(x);
      insertSorted(oppTeamPlayers, player);
    }
    return new GameState(team.getGame().getStatus(),
                         team.getGame().getDisc().getTrajectory(),
                         myTeamPlayers,
                         oppTeamPlayers,
                         team.getGame().getStallCount(),
                         team.getEndzone(),
                         team.getGame().getElapsedTime());
  }

  private void initFlightData() {
    flightDataTable = new Hashtable();
    for (Enumeration throwTypes = ThrowType.elements(); throwTypes.hasMoreElements(); ) {
      ThrowType type = (ThrowType)throwTypes.nextElement();
      double throwHeight = getHeight() * type.getHeight();
```

```
    FlightData flightData;
    flightData =
      new FlightData(Constants.REFRESH_INTERVAL,
                     type.getAngle(),
                    throwHeight,
                     ((Integer)throwPowerTable.get(type)).intValue());
    flightDataTable.put(type, flightData);
  }
}

private void insertSorted(PlayerInfo team[], Player player) {
  if (team[0] == null) {
    team[0] = getPlayerInfo(player);
  }
  else {
    for(int x=0;x<team.length;x++) {
      if (team[x] == null) {
        team[x] = getPlayerInfo(player);
        return;
      }
      else {
        if (player.position.x < team[x].getPosition().x) {
          shift(team, x);
          team[x] = getPlayerInfo(player);
          return;
        }
        else if (player.position.x == team[x].getPosition().x) {
          if (player.position.y < team[x].getPosition().y) {
            shift(team, x);
            team[x] = getPlayerInfo(player);
            return;
          }
        }
      }
    }
  }
}

private void shift(PlayerInfo team[], int index) {
  for(int x=team.length-1;x>=index+1;x--) {
    team[x] = team[x-1];
  }
}

/**
 * Return an estimate of the position of the game disc.
 * @param time the amount of time to estimate for, in milliseconds
 * @return a Point3d object representing the estimated position of the disc.
 */
protected final Point3d predictDiscLocation(double time) {
  Disc disc = team.getGame().getDisc();
  Point3d pt = disc.predictLocation(time);
  return pt;
}

/**
 * Return how long it takes for a player to run to a point.
 * The function takes into account the player's current velocity, and
 * assumes that the player would run at a straight line to the destination.
 * @param dest the point to run to
 * @return the time to run to the point, in milliseconds
 */
protected final double timeToPoint(Point3d dest) {
  Vector3d newVelocity = new Vector3d(dest.x - position.x, dest.y - position.y, 0);
  double newSpeed;
  if (velocity.equals(Constants.ZERO_VELOCITY)) {
    newSpeed = 0;
  } else {
    double angle = velocity.angle(newVelocity);
    newSpeed = velocity.length() * (1 - pctLoss(angle));
  }
```

```
    double distance = dest.distance(position);
    double finalTime = 0;
    if (newSpeed < burstSpeed) {
      double time = (burstSpeed - newSpeed) / burstAcc;
      double tempDist = newSpeed * time + 0.5 * burstAcc * time * time;
      if (tempDist >= distance)
        return (-newSpeed + Math.sqrt(newSpeed * newSpeed + 2 * burstAcc * distance)) /
burstAcc;
      else {
        distance -= tempDist;
        newSpeed = burstSpeed;
        finalTime = Constants.BURST_TIME / 1000D;
      }
    }

    if (newSpeed >= burstSpeed && newSpeed < topSpeed) {
      double time = (topSpeed - newSpeed) / afterAcc;
      double tempDist = newSpeed * time + 0.5 * afterAcc * time * time;
      if (tempDist >= distance)
        return finalTime + (-newSpeed + Math.sqrt(newSpeed * newSpeed + 2 * afterAcc *
distance)) / afterAcc;
      else {
        distance -= tempDist;
        newSpeed = topSpeed;
        finalTime = accelTime / 1000D;
      }
    }

    if (newSpeed == topSpeed) {
      return finalTime + distance / topSpeed;
    }

    return 0;
  }

  protected final double getMinThrowTime(ThrowType type, double dist) {
    FlightData data = (FlightData)flightDataTable.get(type);
    return data.getMinTime(dist);
  }

  protected final double getMaxThrowTime(ThrowType type, double dist) {
    FlightData data = (FlightData)flightDataTable.get(type);
    return data.getMaxTime(dist);
  }

  /**
   * Returns the speed needed for the player to hit the target
   * with the specified throw in the specified time.
   * @param target the point to throw to
   * @param throwType the type of throw
   * @param time to time alloted to reach target
   */
  protected final Vector2d getThrowSpeed(Point3d target, ThrowType throwType, double
time) {
    Point3d discPosition = team.getGame().getDisc().getPosition();
    double distance = PointMath.distance2d(discPosition, target);
    FlightData flightData = (FlightData)flightDataTable.get(throwType);
    double speed = flightData.getSpeed(distance, time);
    Vector3d v = new Vector3d((Tuple3d)target);
    v.sub((Tuple3d)discPosition);
    v.normalize();
    v.scale(speed);
    return new Vector2d(v.x, v.y);
  }

  /**
   * Calls the player's decision function, and then updates the player's
   * state.
   * @param inc the increment of time, in milliseconds
   */
  public final void action(double inc) {
```

```
  if (!specialState) {
    decide(inc);
  }
  update(inc);
}

public final void updateCollisionPosition() {
  position = team.getGame().getCollisionPosition(this);
}

/**
 * Decides what the player should do in the given increment.
 * @param inc the increment of time, in milliseconds
 */
public final void decide(double inc) {

  GameStatus status = team.getGame().getStatus();
  if (status != GameStatus.PRE_PULL) {
    if (team.hasPossession()) {
      offense(inc);
    }
    else {
      defense(inc);
    }
  }

}

/**
 * Abstract method that decides what the player should do in the
 * given increment if the player is on offense.
 * @param inc the increment of time, in milliseconds
 */
public abstract void offense(double inc);

/**
 * Abstract method that decides what the player should do in the
 * given increment if the player is on defense.
 * @param inc the increment of time, in milliseconds
 */
public abstract void defense(double inc);

public final Point3d predictLocation(double time) {
  InnerPlayer p = new InnerPlayer(this);
  while (time > 0) {
    p.update(Math.min(time, Constants.REFRESH_INTERVAL));
    time = time - Constants.REFRESH_INTERVAL;
  }
  return p.getPosition();
}

/**
 * Updates the coordinates of @param disc3D so that it is displayed
 * correctly by World3D.
 */
public void updateGraphics() {
  boolean hopping = false;
  double hopHeight = 0;
  if (hopping) {
    hopHeight = Math.pow(Math.sin(position.y * Math.PI/2), 2);
  }

  double xLocation3D =
    position.x / Constants.GRAPHICS_3D_CONVERSION;
  double yLocation3D =
    position.y / Constants.GRAPHICS_3D_CONVERSION;
  double zLocation3D =
    (hopHeight + position.z) / Constants.GRAPHICS_3D_CONVERSION;

  Point3d location = new Point3d(xLocation3D, yLocation3D, zLocation3D);
  player3D.setLocation(location, direction - Math.PI/2);
```

```
}


/**
 * Updates the player's position and velocity in the given increment.
 * @param inc the increment of time, in milliseconds
 */
public final void update(double inc) {
  Point3d oldPosition = new Point3d(position);
  Point3d dest = getDestination();
  if (dest != null) {
    if (!dest.equals(position)) {
      /**
         if the action takes place entirely in burst time or after burst time,
         run this
      **/
      if (time >= Constants.BURST_TIME || time + inc <= Constants.BURST_TIME) {
        double avgSpeed, currSpeed, finalSpeed;
        Point3d tempDest;
        Vector3d tempVel;

        currSpeed = velocity.length();
        finalSpeed = calcSpeed(inc);
        avgSpeed = (currSpeed + finalSpeed) / 2D;

        tempVel = getVelocity(position, dest, avgSpeed);
        tempDest = new Point3d(position.x + tempVel.x * inc / 1000D,
                               position.y + tempVel.y * inc / 1000D,
                               0);
        /**
           check if the destination the player would be running to is
           past the player's travel point. if so, set the player's
           position to the travel point. if the player has an additional
           travel point, set his velocity based on the angle of the
           cut, and then call update again.
        **/
        if (PointMath.inBetween(dest, position, tempDest)) {
          // this is a linear approximation, suitable for small increments
          double inc1;
          if (dest.x == position.x) {
            inc1 = (dest.y - position.y) / velocity.y;
          }
          else {
            inc1 = (dest.x - position.x) / velocity.x;
          }
          finalSpeed = calcSpeed(inc1);
          avgSpeed = (currSpeed + finalSpeed) / 2D;
          velocity = getVelocity(position, dest, finalSpeed);
          position = dest;
          travel.remove(0);
          time = 0;
          Point3d newDest = getDestination();
          if (newDest != null) {
            if (!newDest.equals(position)) {
              Vector3d newDirection = new Vector3d(newDest.x - position.x, newDest.y -
position.y, 0);
              double angle = velocity.angle(newDirection);
              double newSpeed = velocity.length() * (1 - pctLoss(angle));
              if (newSpeed < burstSpeed) {
                time = newSpeed / burstSpeed * Constants.BURST_TIME;
              }
              else {
                time = Constants.BURST_TIME + 50;
              }
              newDirection.normalize();
              newDirection.scale(newSpeed);
              velocity = newDirection;
            } else {
              travel.remove(0);
            }
```

```
          } else {
            velocity = Constants.ZERO_VELOCITY;
            time = 0;
            if (specialState) {
              System.out.println("leaving special state");
              specialState = false;
              team.getGame().startStallCount();
            }
          }
          update(inc - inc1);
        }
        else {
          time += inc;
          velocity = getVelocity(position, dest, finalSpeed);
          position = tempDest;
        }
      }
      /**
         if the increments spans burst time and after-burst time, split
         the update into 2 parts
      **/
      else {
        double inc1 = Constants.BURST_TIME - time;
        double inc2 = inc - inc1;
        update(inc1);
        update(inc2);
      }
    } else {
      travel.remove(0);
    }
  } else {
    velocity = Constants.ZERO_VELOCITY;
    time = 0;
    if (specialState) {
      specialState = false;
      team.getGame().startStallCount();
    }
  }

  // update direction player is facing
  if (hasDisc) {
    direction = getDefaultDirection();
    updateStationaryDiscPosition();
  }
  else if (!position.equals(oldPosition)) {
    direction = PointMath.getXYAngle(oldPosition, position);
  }
  else {
    direction = PointMath.getXYAngle(position, team.getGame().getDisc().getPosition());
  }


}

private void setPhysicalConstants() {
  // find the player's top speed, based on his rating
  double scale = (double)skillSet.getSkill(SkillType.Speed) / 10D;
  double delta = Constants.TOP_SPEED - Constants.BOT_SPEED;
  topSpeed = Constants.BOT_SPEED + scale * delta;

  // find the player's acceleration time, based on his rating
  scale = (10D - (double)skillSet.getSkill(SkillType.Acceleration)) / 10D;
  delta = Constants.BOT_ACCEL - Constants.TOP_ACCEL;
  accelTime = Constants.TOP_ACCEL + scale * delta;

  // find the player's rate of acceleration during burst
  burstSpeed = Constants.BURST_PCT * topSpeed;
  burstAcc = burstSpeed / (Constants.BURST_TIME / 1000D);

  // find the player's rate of acceleration after the burst
  double timeLeft = accelTime - Constants.BURST_TIME;
```

```
  afterAcc = (topSpeed - burstSpeed) / (timeLeft / 1000D);

  // find the player's maximum percentage loss of speed after a 180-degree cut
  scale = (10D - (double)skillSet.getSkill(SkillType.CutAbility)) / 10D;
  delta = Constants.BOT_PCT - Constants.TOP_PCT;
  maxPctLoss = Constants.TOP_PCT + scale * delta;

  // compute the player's throwing power
  throwPowerTable = new Hashtable();
  int power = skillSet.getSkill(SkillType.BackhandPower);
  throwPowerTable.put(ThrowType.LOW, new Integer(3*power));
  throwPowerTable.put(ThrowType.NORMAL, new Integer(5*power));
  throwPowerTable.put(ThrowType.HUCK, new Integer(5*power));

  // compute player's reaction time
  int rt = skillSet.getSkill(SkillType.ReactionTime);
  reactionTime = (11-rt) * Constants.REFRESH_INTERVAL;
}

/**
 * Returns a player's velocity after he has cut with the specified angle.
 * @param angle the angle, in radians, that the player has cut, between 0 and pi
 */
private double pctLoss(double angle) {
  return maxPctLoss * angle / Math.PI;
}

/**
 * Returns the proper velocity vector given the origin, destination,
 * and velocity magnitude.
 **/
private Vector3d getVelocity(Point3d orig, Point3d dest, double velMag) {
  // ignore the z-component of the origin and destination
  Vector2d vect = new Vector2d(dest.x - orig.x, dest.y - orig.y);
  vect.normalize();
  vect.scale(velMag);
  return new Vector3d(vect.x, vect.y, 0);
}

/**
 * Calculate what the speed of a player would be after an increment.
 * @param inc the amount of time to increment
 */
private double calcSpeed(double inc) {
  if (time >= Constants.BURST_TIME)
    return Math.min(topSpeed, velocity.length() + afterAcc * inc / 1000D);
  else
    return velocity.length() + burstAcc * inc / 1000D;
}

/**
 * The TravelPoint class stores three pieces of information.
 * Point stores where the desired destination is
 * timestamp stores when the point becomes valid
 * remove is a boolean that stores whether the travel point is still wanted.
 * If remove is true, then the point will be removed the first time it is
 * executed, regardless of whether it has been used.
 */
private class TravelPoint {
  private Point3d point;
  private double timestamp;
  private boolean used;
  private boolean remove;

  public TravelPoint(Point3d point, double timestamp) {
    this.point = point;
    this.timestamp = timestamp;
    this.remove = false;
    this.used = false;
  }
```

```
    public Point3d getPoint() {
      return point;
    }

    public double getTimeStamp() {
      return timestamp;
    }

    public boolean isRemoved() {
      return remove;
    }

    public void setRemove() {
      remove = true;
    }

    public boolean isUsed() {
      return used;
    }

    public void setUsed() {
      used = true;
    }

  }

  // used for predicting player movement
  private class InnerPlayer extends Player {
    public InnerPlayer(Player player) {
      super(player);
    }

    public void resetOffense() {
      return;
    }

    public void resetDefense() {
      return;
    }

    public void offense(double inc) {
      return;
    }

    public void defense(double inc) {
      return;
    }

  }

}
```

The source that follows is extracted from an implementing subclass of Player, and is the core of the offensive portion of the player.

```
  public void offense(double time) {
    gs = getGameState();
    if (gs.getStatus() == GameStatus.PULL) {
      if (!gs.getDiscInfo().getVelocity().equals(Constants.ZERO_VELOCITY)) {
        if (!doOnce) {
          /**
           * if best handler, go get the disc, otherwise,
           * line up in the stack
           */
          setPivot(predictDiscLocation(10000));
          myPivot = predictDiscLocation(10000);
          if (bestHandler()) {
            System.out.println(getId() + " is best handler. pivot is " + pivot);
            setDestination(pivot);
```

```
      } else {
        setDestination(normalStackPosition(pivot));
      }
      doOnce = true;
    }
  }
} else if (gs.getStatus() == GameStatus.PLAY) {
  doOnce = false;
  Trajectory discInfo = gs.getDiscInfo();
  /**
   * always try to catch the disc
   */
  if (catchDisc()) {
    if (Constants.inEndzone(getPosition())) {
      setPivot(Constants.closestEndzonePosition(discInfo.getPosition()));
    } else {
      setPivot(getPosition());
    }
  } else {
    if (discInfo.getPosition().z == 0D &&
        discInfo.getVelocity().equals(Constants.ZERO_VELOCITY) &&
        !someoneHasDisc()) {
      if (bestHandler()) {
        if (!pickup()) {
          if (!doPickupOnce) {
            setDestination(discInfo.getPosition());
          }
        }
      }
      if (!doPickupOnce) {
        Point3d newPivot;
        if (Constants.inEndzone(discInfo.getPosition())) {
          newPivot = Constants.closestEndzonePosition(discInfo.getPosition());
        } else {
          newPivot = new Point3d(discInfo.getPosition());
        }
        setPivot(newPivot);
        myPivot = newPivot;
        if (!bestHandler()) {
          setDestination(normalStackPosition(newPivot));
        }
      }
      doPickupOnce = true;
    } else {
      doPickupOnce = false;
      if (hasDisc()) {
        setPivot(getPosition());
        if (gs.stallCount() < (Constants.STALL_COUNT - 2000)) {
          double besty;
          double bestTime = 0D;
          int bestindex = -1;
          if (gs.getEndzone() == Constants.NORTH) {
            besty = -100;
          } else {
            besty = 100;
          }

          PlayerInfo myTeam[] = gs.getMyTeamInfo();

          /**
           * only consider non-moving players if they are
           * really poached.
           */
          for(int x=0; x<myTeam.length; x++) {
            if (myTeam[x].getVelocity().equals(Constants.ZERO_VELOCITY)) {
              Point3d target = myTeam[x].getPosition();
              double minTime = getMinThrowTime(ThrowType.NORMAL,
                                               getPosition().distance(target));
              if (openFactor(myTeam[x], minTime) > 5) {
                if (deeper(besty, target.y)) {
                  besty = target.y;
```

```
            bestindex = x;
          }
        }
      }
    }

    if (bestindex == -1) {
      double maxOpen = 0D;
      if (gs.getEndzone() == Constants.NORTH) {
        besty = -100;
      } else {
        besty = 100;
      }

      for(int x=0; x<myTeam.length; x++) {
        if (!myTeam[x].getVelocity().equals(Constants.ZERO_VELOCITY)) {
          for(double y=500D; y<= 2500D; y+= 100D) {
            Point3d target = myTeam[x].predictLocation(y);
            double distance = getPosition().distance(target);
            double minTime, maxTime;
            if (distance < 10) {
              minTime = getMinThrowTime(ThrowType.LOW, distance);
              maxTime = getMaxThrowTime(ThrowType.LOW, distance);
            } else if (distance >= 10 && distance < 40) {
              minTime = getMinThrowTime(ThrowType.NORMAL, distance);
              maxTime = getMaxThrowTime(ThrowType.NORMAL, distance);
            } else {
              minTime = getMinThrowTime(ThrowType.HUCK, distance);
              maxTime = getMaxThrowTime(ThrowType.HUCK, distance);
            }
            if (y >= minTime && y <= maxTime) {
              if (onLeft(myTeam[x]) || onRight(myTeam[x])) {
                if (openFactor(myTeam[x], y) > maxOpen) {
                  maxOpen = openFactor(myTeam[x], y);
                  bestindex = x;
                  bestTime = y;
                }
              }
            }
          }
        }
      }
      if (maxOpen < 1 && gs.stallCount() > 4000) {
        bestindex = -1;
      }
    }

    if (bestindex != -1) {

      Point3d target = myTeam[bestindex].predictLocation(bestTime);
      ThrowType throwType;
      double distance = getPosition().distance(target);
      if (distance < 10) {
        throwType = ThrowType.LOW;
      } else if (distance >= 10 && distance < 40) {
        throwType = ThrowType.NORMAL;
      } else {
        throwType = ThrowType.HUCK;
      }
      Vector2d speed = getThrowSpeed(target, throwType, bestTime);
      throwDisc(throwType, speed);
    }
  }
} else {
  if (myPivot == null) {
    myPivot = pivot;
  }

  if (!myPivot.equals(pivot)) {
    clearTravelPath();
    isCutting = false;
```

```
          myPivot = pivot;
        }

        if (!isCutting) {
          if (farthestBack()) {
            isCutting = cut();
          }
          if (!isCutting) {
            if (clogging()) {
              setDestination(normalStackPosition(pivot));
            }
          }
        } else {
          if (clogging()) {
            isCutting = false;
            setDestination(normalStackPosition(pivot));
          }
        }
      }
    }
  }
 }
}

// Offense Helper Methods

private boolean deeper(double y, double compy) {
  if (gs.getEndzone() == Constants.NORTH) {
    return compy > y;
  } else {
    return compy < y;
  }
}

private int getMarkerId() {
  double minDistance = 1000;
  int minId = -1;
  PlayerInfo oppTeam[] = gs.getOppTeamInfo();
  for(int x=0; x<oppTeam.length; x++) {
    double distance = getPosition().distance(oppTeam[x].getPosition());
    if (distance < minDistance) {
      minId = oppTeam[x].getId();
      minDistance = distance;
    }
  }
  if (minDistance < 3) {
    return minId;
  } else {
    return -1;
  }
}

private double openFactor(PlayerInfo player, double time) {
  double minOpen;
  int markerId;
  PlayerInfo oppTeam[];
  PlayerInfo marker;
  Point3d discLoc, playerLoc;

  minOpen = 1000;
  discLoc = gs.getDiscInfo().getPosition();
  playerLoc = player.predictLocation(time);
  oppTeam = gs.getOppTeamInfo();
  markerId = getMarkerId();

  for(int x=0; x<oppTeam.length; x++) {
    /**
     * temporary ignoring the mark,
     * check to see how close the player is to the throwing
     * lane between the thrower and the receiver, and also
     * how close the player is to the receiver.
```

```
      */
      Point3d oppPlayerLoc = oppTeam[x].predictLocation(time);
      double openFactor = playerLoc.distance(oppPlayerLoc);
      if (PointMath.inBetween(oppPlayerLoc, discLoc, playerLoc)) {
        if (markerId == oppTeam[x].getId()) {
          double rand = Math.random();
          if (rand > 0.2) {
            openFactor = Math.min(openFactor,
                                  PointMath.distancePointToLine(oppPlayerLoc, discLoc,
                                                                 playerLoc));
          }
        } else {
          openFactor = Math.min(openFactor,
                                PointMath.distancePointToLine(oppPlayerLoc, discLoc,
                                                               playerLoc));
        }
      }
      minOpen = Math.min(minOpen, openFactor);
    }
    return minOpen;
  }

  private void setPivot(Point3d pivot) {
    if (this.pivot == null || !this.pivot.equals(pivot)) {
      System.out.println("pivot being set by " + getId());
      this.pivot = new Point3d(pivot);
      if (myPivot == null) {
        myPivot = new Point3d(pivot);
      }
      if (gs.getEndzone() == Constants.NORTH) {
        bisect = new Vector3d(-pivot.x,
                              (Constants.BASE_FIELD_LENGTH/2 + pivot.y)/2 - pivot.y,
                              0);
      } else {
        bisect = new Vector3d(-pivot.x,
                              (-Constants.BASE_FIELD_LENGTH/2 + pivot.y)/2 - pivot.y,
                              0);
      }
    }
  }

  // clogging here means standing on one side of the field, doing nothing
  private boolean clogging() {
    if ((onSide(getPosition(), true, true) || onSide(getPosition(), false, true))
        && getVelocity().equals(Constants.ZERO_VELOCITY)) {
      return true;
    }
    return false;
  }

  private boolean cut() {
    // check right/left side clog
    // if one or other clogged, cut to the other
    // else, do 50-50 split in cuts
    boolean leftSideClogged = leftSideClogged();
    boolean rightSideClogged = rightSideClogged();
    if (leftSideClogged && !rightSideClogged) {
      cutRight();
      return true;
    } else if (rightSideClogged && !leftSideClogged) {
      cutLeft();
      return true;
    } else if (!rightSideClogged && !leftSideClogged) {
      double chance = Math.random();
      if (chance <= 0.5) {
        cutLeft();
      } else {
        cutRight();
      }
      return true;
    }
```

```
    return false;
  }

  private void cutLeft() {
    Point3d pivotClone = new Point3d(pivot);
    Vector3d bisect = new Vector3d(this.bisect);
    Vector3d cut;
    bisect.normalize();
    bisect.scale(10);
    double rand = Math.random();
    if (rand >= 0.85) {
      Vector3d bisectPerp = new Vector3d(bisect.y, -bisect.x, 0);
      bisect.normalize();
      bisect.scale(40);
      System.out.println(getId() + " is cutting deep left.");
      cut = new Vector3d(bisect.x + bisectPerp.x,
                         bisect.y + bisectPerp.y,
                         0);
    } else {
      System.out.println(getId() + " is cutting in left.");
      cut = new Vector3d(bisect.x + bisect.y,
                         bisect.y - bisect.x,
                         0);
    }
    pivotClone.add(cut);
    if (gs.getEndzone() == Constants.NORTH) {
      pivotClone.set(Math.min(pivotClone.x, Constants.BASE_FIELD_WIDTH/2D),
                     Math.min(pivotClone.y, Constants.BASE_FIELD_LENGTH/2D),
                     0);
    } else {
      pivotClone.set(Math.max(pivotClone.x, -Constants.BASE_FIELD_WIDTH/2D),
                     Math.max(pivotClone.y, -Constants.BASE_FIELD_LENGTH/2D),
                     0);
    }
    setDestination(pivotClone);
  }

  private void cutRight() {
    Point3d pivotClone = new Point3d(pivot);
    Vector3d bisect = new Vector3d(this.bisect);
    Vector3d cut;
    double rand = Math.random();
    bisect.normalize();
    bisect.scale(10);
    if (rand >= 0.85) {
      Vector3d bisectPerp = new Vector3d(-bisect.y, bisect.x, 0);
      bisect.normalize();
      bisect.scale(40);
      cut = new Vector3d(bisect.x + bisectPerp.x,
                         bisect.y + bisectPerp.y,
                         0);
    } else {
      cut = new Vector3d(bisect.x - bisect.y,
                         bisect.x + bisect.y,
                         0);
    }
    pivotClone.add(cut);
    if (gs.getEndzone() == Constants.NORTH) {
      pivotClone.set(Math.max(-Constants.BASE_FIELD_WIDTH/2D, pivotClone.x),
                     Math.max(-Constants.BASE_FIELD_LENGTH/2D, pivotClone.y),
                     0);
    } else {
      pivotClone.set(Math.min(Constants.BASE_FIELD_WIDTH/2D, pivotClone.x),
              Math.min(Constants.BASE_FIELD_LENGTH/2D, pivotClone.y),
              0);
    }
    setDestination(pivotClone);
  }

  private boolean leftSideClogged() {
    return sideClogged(true);
```

```
}

private boolean rightSideClogged() {
  return sideClogged(false);
}

private boolean sideClogged(boolean isLeft) {
  if (onSide(getPosition(), isLeft, true) && onSide(getVelocity(), isLeft, false)) {
    return true;
  } else {
    PlayerInfo myTeam[] = gs.getMyTeamInfo();
    for(int x=0;x<myTeam.length;x++) {
      if (!myTeam[x].hasDisc()) {
        if (isLeft && onLeft(myTeam[x])) {
          return true;
        } else if (!isLeft && onRight(myTeam[x])) {
          return true;
        }
      }
    }
  }
  return false;
}

private boolean onLeft(PlayerInfo player) {
  return onSide(player.getPosition(), true, true) &&
    onSide(player.getVelocity(), true, false);
}

private boolean onRight(PlayerInfo player) {
  return onSide(player.getPosition(), false, true) &&
    onSide(player.getVelocity(), false, false);
}

// left and right is from the perspective of the thrower
private boolean onSide(Tuple3d pos, boolean isLeft, boolean isPos) {
  double slope = bisect.y/bisect.x;
  double offset = -slope * pivot.x + pivot.y;

  Point3d posClone = new Point3d(pos);
  if (!isPos) {
    posClone.add(pivot);  // offset if velocity
  }

  double compx = slope * posClone.x + offset;
  if (Math.abs(compx - posClone.y) < 0.001) {
    return !isPos;
  }
  if (isLeft) {
    if (gs.getEndzone() == Constants.NORTH) {
      if (slope > 0) {
        return posClone.y < compx;
      } else {
        return posClone.y > compx;
      }
    } else {
      if (slope > 0) {
        return posClone.y > compx;
      } else {
        return posClone.y < compx;
      }
    }
  } else {
    if (gs.getEndzone() == Constants.NORTH) {
      if (slope > 0) {
        return posClone.y > compx;
      } else {
        return posClone.y < compx;
      }
    } else {
      if (slope > 0) {
```

```
        return posClone.y < compx;
      } else {
        return posClone.y > compx;
      }
    }
  }
}

// only accounts for non-moving people
private boolean farthestBack() {
  if (!getVelocity().equals(Constants.ZERO_VELOCITY)) {
    return false;
  } else {
    double farthest = getPosition().y;
    PlayerInfo myTeam[] = gs.getMyTeamInfo();
    for(int x=0;x<myTeam.length;x++) {
      if (myTeam[x].getVelocity().equals(Constants.ZERO_VELOCITY) &&
          !myTeam[x].hasDisc() &&
          fartherBack(myTeam[x].getPosition())) {
        return false;
      }
    }
    return true;
  }
}

private boolean fartherBack(Point3d position) {
  if (gs.getEndzone() == Constants.NORTH) {
    return getPosition().y < position.y;
  }
  else {
    return getPosition().y > position.y;
  }
}

private double getHandleSkill(PlayerSkill skillSet) {
  return ((double)skillSet.getSkill(SkillType.ForehandAcc) +
          0.5 * (double)skillSet.getSkill(SkillType.ForehandPower)) *
    ((double)skillSet.getSkill(SkillType.BackhandAcc) +
     0.5 * (double)skillSet.getSkill(SkillType.BackhandPower));
}

private double distanceToDisc(Point3d location) {
  return location.distance(gs.getDiscInfo().getPosition());
}

private boolean bestHandler() {
  return getId() == bestHandlerId();
}

private int bestHandlerId() {
  double bestHandle = getHandleSkill(getSkills());
  int bestId = getId();
  PlayerInfo myTeam[] = gs.getMyTeamInfo();
  for(int x=0;x<myTeam.length;x++) {
    double compHandle = getHandleSkill(myTeam[x].getSkills());
    if (compHandle > bestHandle) {
      bestId = myTeam[x].getId();
      bestHandle = compHandle;
    } else if (compHandle == bestHandle && myTeam[x].getId() < bestId) {
      bestId = myTeam[x].getId();
    }
  }
  return bestId;
}

private int idRank() {
  // get the player's rank by Id
  // the player that is excluded is a) the one with the disc
  // b) otherwise the best handler chasing after the disc
```

```
  PlayerInfo myTeam[] = gs.getMyTeamInfo();
  boolean hasDisc = false;
  boolean excludeHandler = true;
  int hasDiscId = -1;
  int myRank = 1;
  for(int x=0;x<myTeam.length;x++) {
    if (myTeam[x].hasDisc()) {
      hasDisc = true;
      hasDiscId = myTeam[x].getId();
      excludeHandler = false;
    }
  }

  for(int x=0;x<myTeam.length;x++) {
    // if the player has the disc, exclude
    // if the hasDisc is false and player is best handler, exclude
    if (!((myTeam[x].hasDisc()) ||
          (!hasDisc && myTeam[x].getId() == bestHandlerId()))) {
      if (myTeam[x].getId() < getId()) {
        myRank++;
      }
    }
  }
  return myRank;
}

private Point3d normalStackPosition(Point3d pivot) {
  Point3d pivotClone = new Point3d(pivot);
  int rank = idRank();
  Vector3d bisect = new Vector3d(this.bisect);
  bisect.normalize();
  double distance = 10 + rank * 5;
  bisect.scale(distance);
  pivotClone.add(bisect);
  return pivotClone;
}

private Point3d closestStackPosition(Point3d pivot) {
  Point3d pivotClone = new Point3d(pivot);
  return new Point3d();
}
```

Bibliography

[1] Michael van Lent, John Laird, et al. *Intelligent Agents in Computer Games.* Artificial Intelligence Lab, University of Michigan

[2] Glenn Elert, editor. *The Physics Factbook: Speed of the Fastest Human Running.* http://hypertextbook.com/facts/2000/KatarzynaJanuszkiewicz.shtml

[3] Dean Bolton. *Three-Dimensional Java Ultimate Coaching Simulator.* Master's thesis, Massachusetts Institute of Technology, 2002.

[4] David Bailey. *Intelligent Three-Dimensional Ultimate Frisbee Simulator.* Master's thesis, Massachusetts Institute of Technology, 2003

[5] Ultimate Players Association. *The Official Rules of Ultimate – 10th Edition.* http://www.upa.org/ultimate/rules/10thfinl.pdf