Three-Dimensional Ultimate

Coaching Simulator

by

Dean Bolton

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

September 16, 2002

Author_____

Department of Electrical Engineering and Computer Science
September 16, 2002

Certified
by_____

Seth Teller
Thesis Supervisor

Accepted
by_____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Three-Dimensional Ultimate Coaching Simulator
by
Dean Bolton

Submitted to the
Department of Electrical Engineering and Computer Science

September 16, 2002

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

ABSTRACT

The ultimate coaching simulator is a tool for anyone interested in the sport of ultimate to model an actual game. The simulator allows a user to view the progression of a game from multiple angles in a three-dimensional environment. The main panel of the simulator contains the three-dimensional environment, which can be freely rotated using the mouse. Additional panels are accessible via the menu bar. The field control panel is a two-dimensional overhead representation of the game. The simulation control panel allows one to easily pause a game in progress, as well as rewind the progression in one-second intervals. The player and team control panels provide information on the selected player and/or team as well as controls for modifying the individual variables. The coaching simulator is available as an applet via a web browser; however, it can also be run locally as an application.

Thesis Supervisor: Seth Teller
Title: Associate Professor, MIT Laboratory for Computer Science

Contents

Figures

1 The Sport of Ultimate

Ultimate is a non-contact disc sport played by two teams of seven players [UPA 2001]. An official game of Ultimate is played on a field one hundred ten meters long by thirty-seven meters wide. There are two end zones, which are twenty-three meters deep and thirty-seven meters wide, at either end of the playing field. Games are usually played to fifteen points with a halftime after one team scores eight points.

Each point is started by the teams lining up on the front line of the end zone that they are defending. The team that just scored pulls (throws) the disc to the opposing team. A goal is scored when a player catches any legal pass in the end zone that player is attacking. Players are not allowed to run while holding the disc. The disc is advanced by throwing or passing it to other players. The disc may be passed in any direction. Any time a pass is incomplete, intercepted, knocked down, or contacts an out-of-bounds area, a turnover occurs, resulting in an immediate change of the team in possession of the disc [UPA 2001]. Substitutions are only allowed in between points or if a player becomes injured.

Unlike many other popular sports, the rules of Ultimate are enforced by the players themselves. The utmost rule of Ultimate is the Spirit of the Game, a spirit of sportsmanship that places the responsibility for fair play on the player himself [UPA 2001]. The rules of Ultimate are set up to describe the way the game is to be played. There are no harsh penalties for violations of the rules because intentional cheating is

against the Spirit of the Game. Instead, the rules describe methods to resume play after inadvertent violations.

The Ultimate Players Association (UPA) is the main governing body of the sport in the US. With over 88,000 members, the UPA exists to promote and support the sport of Ultimate, as well as organize and conduct national competition. The World Flying Disc Federation (WFDF) is the international organization that works in conjunction with the UPA to organize international competitions. The WFDF Championships are held on a biannual basis and gather participants from more than thirty (30) nations.

2 Introduction

The scope of this project was to make an ultimate simulator that could be used as a coaching tool. The simulator had to be easy to use as well as easy to distribute to a wide audience. The goal of the simulator was for it to be useable by anyone with access to the Internet. It also had to be robust enough so that novices who had never thrown a disc, as well as players with twenty years of experience, would find the simulator useful.

In order to allow a wide audience access to the simulator, it was decided that the simulator would be developed as a Java applet/application. That way anyone could view the simulator with a web browser or download the application to run locally. The release of the Java™ 2 environment and the Java 3D™ API allowed for the development of platform-independent code. The Java™ 2 plug-in is an upgrade of the Java™ Virtual Machine included with most browsers that allows viewing of applets that make use of Java 3D™.

3 Implementation

There were two main considerations when implementing the simulator. First, the simulator should be as feature-rich as possible in order to make it useful to the widest audience. Second, the implementation should be a good foundation that allows for future improvement. For example, there are many different levels of offense at which a player can operate. Beginners usually run all over the field without consideration for other players. However, advanced players usually know the location of the other thirteen people, as well as the direction in which each of those other thirteen people is moving. Therefore, the implementation should allow for different levels of players that all follow the rules of the game.

3.1 Subsumption Architecture

A subsumption architecture is used to implement the players' artificial intelligence [Brooks 1985]. The idea of a subsumption architecture is to build varying layers of control. The fundamental layer, or level 0 layer, controls the basic movements of the player, such as not running with the disc, never running into other players, staying in bounds and the other fundamental rules of the game. With this level 0 layer, the game is played at a rudimentary level. The players will have no concept of an offensive or defensive strategy; however, the game proceeds according to the rules.

In order to make the players more lifelike, further control layers can be implemented. In a subsumption architecture, each additional layer provides a new level of sophistication to the simulator. The level 0 layer is responsible for making sure that the rules of the game are followed. The level 1 layer can implement different offensive and defensive strategies in order to make the simulation more realistic. However, the level 0 layer still retains control to ensure that the rules of the game are followed. As shown in Figure 1, the basic player class can be extended to implement advanced players who use different guidelines for playing the game.

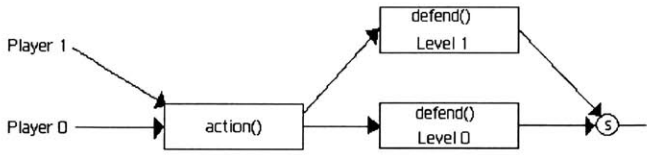

Figure 1: Basic Subsumption Architecture

All extensions of the player class call the same action method that is responsible for adhering to the rules of the game. However, if the player is in *DEFEND* mode, then the action method calls the defend method. For two different levels of players, these could be overriding methods. Therefore, the level 0 player would use the level 0 defend method. However, the level 1 player would subsume the level 0 defend method and use the level 1 defend method.

3.2 Graphical User Interface

Since the simulator is intended to provide a great deal of flexibility, the graphical user interface (GUI) has to incorporate many different subcomponents of the simulator. However, since the target audience is novice and advanced in both the game of Ultimate and the complexities of the computer, the GUI has to be easy to use as well. Therefore,

the GUI is divided into five main panels: the camera control panel, the simulation control panel, the field control panel, the player control panel, and the main panel.

The main panel, shown in Figure 2, is the heart of the simulator. The main panel is composed of two different parts, the menu bar and the three-dimensional display. The menu bar has controls for accessing all of the other panels. The flow of the simulator can also be controlled via the *Simulation* menu. In the applet mode, this is the only panel that appears by default. By dragging the mouse through the three-dimensional window while the left mouse button is depressed, the environment can be rotated around the mouse point. Conversely, dragging the mouse while the right mouse button is depressed results in a scroll of the environment in the dragged direction.

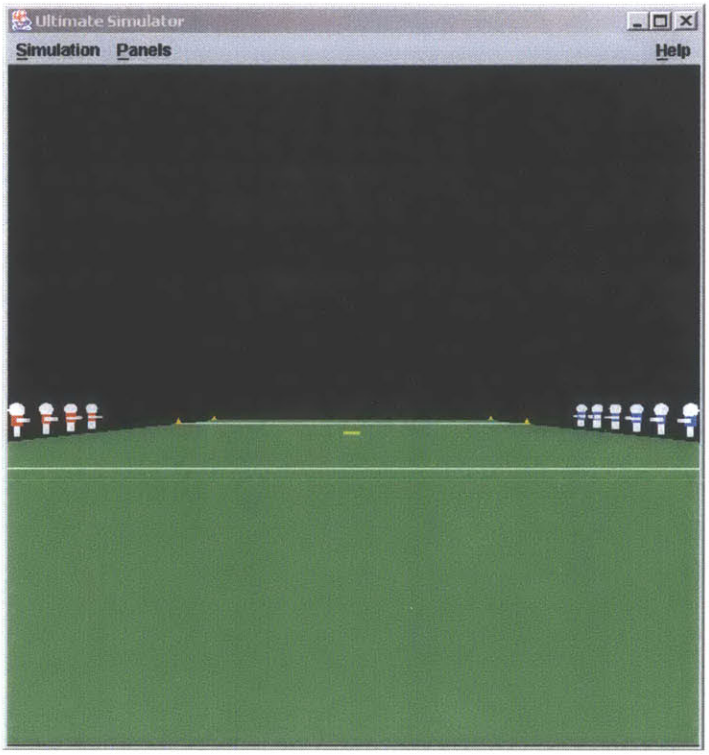

Figure 2: Main Simulator Panel

The player and team control panels are similar in nature. They both can be used to modify private variables of a particular player or team. The player control panel has drop-down boxes to change a player's speed, jumping ability, strength, dexterity, and stamina. These are all private variables in the Player class; however, not all of the variables are related to the action of the player. In the level 0 layer, the player only

factors in the speed variable when moving around on offense and defense. However, extended implementations of the Player class could easily incorporate the other variables into the movement selection process of the player.

Augmenting the main panel, the field control panel contains a two-dimensional representation of the game. As shown in Figure 3, the two-dimension view is the game in its current state from a vantage point directly above the center of the field looking down. The two-dimensional field serves as a backup to the three-dimensional panel to follow the progress of the game. It also provides drag-and-drop functionality for moving players around on the field. By double-clicking on an individual player, the player control panel appears and displays the relevant information about the selected player. Also included in the field control panel is a simple display of the relevant game settings. This display includes the score, the person with the disc, and the current stall count.

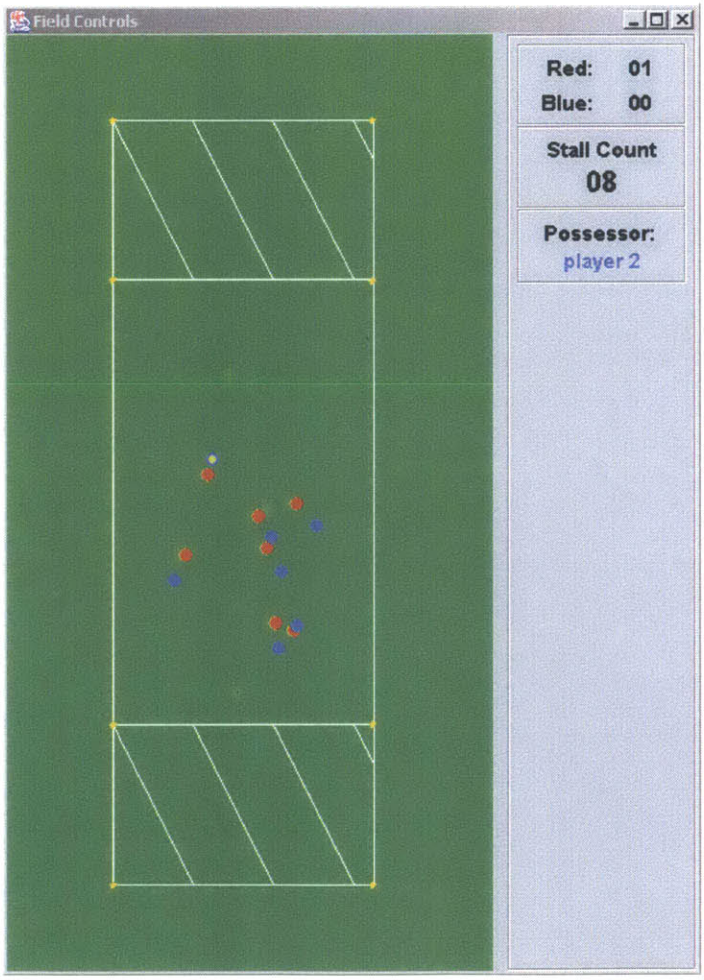

Figure 3: Field Control Panel

The simulation control panel is a graphical representation of the *Simulation* menu in the main panel. Figure 4 shows the panel with the six buttons to control the flow of the simulation. From left to right, the buttons are full rewind, step backward, play, pause, step forward, and full forward. At any point, the simulation may be paused and rewound or advanced. The replay controller, described later, takes a snapshot of the game in one-second intervals. The simulation control panel allows the navigation of these snapshots. The user can click step backward five times to move the simulation back five seconds. Then the user can select play to start the simulation from that point, or the step forward button could be selected to advance the simulation one second. Because the simulations are not pre-determined, the step forward and full forward buttons have no impact unless the simulation has been moved backward in time.

Figure 4: Simulation Control Panel

The last panel in the GUI is the camera control panel, Figure 5. This panel has multiple radio buttons for switching the camera angle in the three-dimensional display of the main panel. This panel is extremely useful if the user has rotated or scrolled the display using the mouse because the custom mouse control tends to result in an unnatural viewpoint. The camera panel allows an easy interface to reset the viewpoint in the three-dimensional panel to a known place.

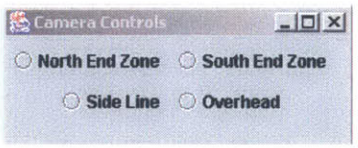

Figure 5: Camera Control Panel

3.3 Player

The Player class is used to control the movement as well as the display of the individual players. The Player class has three core components as diagrammed in Figure 6. The mode component controls whether the player is on offense or defense, as well as what to do when the player has the disc, what to do during the pull, etc. The mode component is stored as an integer called *command*. Each time the player is refreshed, the *action* method is called. This method checks the command against static integers in order to determine what action the player is supposed to take. If the command is *STOP*, the player takes no action for that cycle. If the command is *MOVE_TO*, the player is trying to move from its current location to a specified destination, which is another internal variable. If the player has reached the destination, the command is set to *STOP*; otherwise, the player moves toward the destination and remains in the *MOVE_TO* mode.

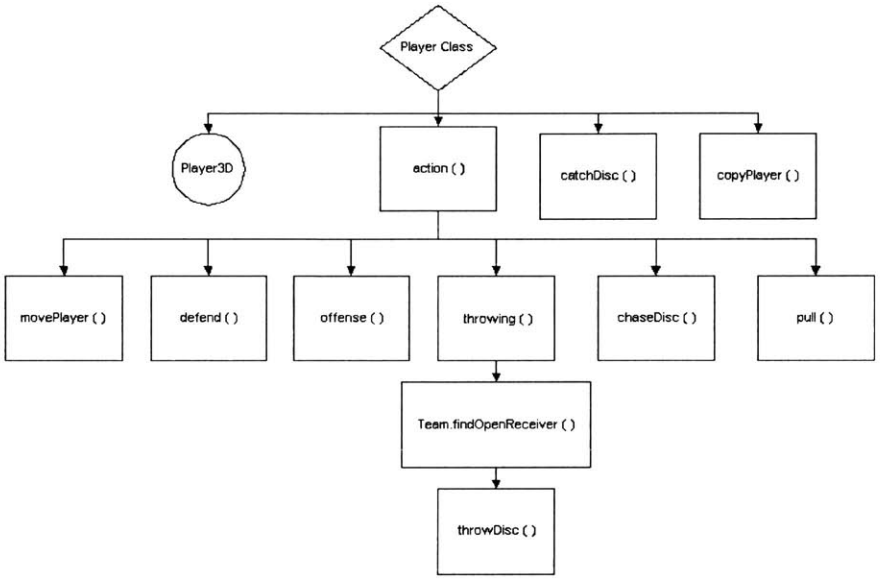

Figure 6: Player Class

If the player is in *DEFEND* mode, the *action* method calls the *defend* method. The entire *defend* method could have been written in the corresponding section of the

action method; however, an extended player would have to re-implement the entire *action* method. By having a separate method call, the extended player simply implements a new defensive strategy, which subsumes the lower level defensive strategy. In the level 0 strategy, the defensive player simply follows his marked player around the field. There are slight delays so that the defensive player is reacting to the offensive player, and a buffer exists so that the defensive player does not run into the player on offense.

Like the *DEFEND* mode, if the player is in the *OFFENSE* mode, a separate call is made to the *offense* method. This allows the implementation of new offensive strategies without rewriting the base functionality of the player. In the level 0 strategy, the player randomly chooses a point in between the disc and the back of the end zone. After waiting a random amount of time between zero and five seconds, the player will move towards the selected destination. This will continue until the player is notified that it has the disc or that there has been a change of possession.

If the player is in possession of the disc, it will be in the *THROW* mode. As with the previous two modes, this mode leads to a separate *throw* method call. In the level 0 implementation, the player will selected a random time before the stall count runs out to throw the disc. The player calls the method *findOpenReceiver* from the Team class in order to determine which of the other six players is the best receiver. Then the player calls the *throwDisc* method in order to pass the disc to the open receiver. After throwing the disc, the player enters the *POST_THROW_OFFENSE* mode. The *POST_THROW_OFFENSE* mode is simply a transition mode that selects a destination for the player that just threw the disc in order to transition into the *OFFENSE* mode.

The final two modes, *PULL* and *CHASE DISC*, are simply transitions in order to continue with the flow of the game. The *PULL* mode is used to select a destination for the pull. The *CHASE_DISC* mode is used to initially pick up the disc after a pull or after a change in possession.

The other core components of the Player class are the *copyPlayer* method and the *Player3D* variable. The *copyPlayer* method is used by the ReplayController to store all of the necessary information about that particular player. The vital requirements, such as location, destination, command mode, and the *Player3D* information, are cloned and then stored as needed by the ReplayController.

The *Player3D* variable is responsible for the three-dimensional representation of the player. It stores the location, as well as the destination of the player, in order to update the three-dimensional window. Since the three-dimensional window is simply a display for the current state, the command mode as well as the physical characteristics, such as speed, dexterity, etc., are not needed in the *Player3D* class. Therefore, updating the rendering of the players in the three-dimensional window is completely contained within the *Player3D* class.

3.3 Team

The *Team* class is responsible for the group movement and actions in the game. As in a real game of Ultimate, the player adheres to the basic rules of the game and has knowledge about how to play defense and how to throw the disc. However, there is a strong team concept in a game of Ultimate that relies on players trusting each other to

behave in a certain manner. For instance, setting a mark in one direction or forming a stack are two basic team concepts. These fundamental team concepts are included in the *Team* class.

The *Team* class is simply an array of players along with variables for the score, for which team is in possession of the disc, for which end zone the team is attacking, and an array of locations that represent where the players line up for the pull. The *Team* class has four important methods: *changePossession*, *pull*, *findOpenReceiver*, and *setMarkedPlayers*.

The *changePossession*, *pull*, and *setMarkedPlayers* are all fundamental methods that make up the level 0 layer of the team architecture. *changePossession* occurs whenever there is a turnover and sets the command mode of the players previously on offense to *DEFEND* as well as putting the players previously on defense into the *OFFENSE* mode. It also selects the nearest player on offense to pick up the disc and begin play if the pass was incomplete. *pull* is a similar method that instructs the pulling team to kickoff. It also selects the nearest player of offense to pick up the disc and begin the game. s*etMarkedPlayers* assigns a player on the other team for each player to defend. In the current implementation, the players simply match up with their counterpart on the other team as indicated by their position in the players array in the *Team* class.

findOpenReceiver is called by the player with the disc when it decides to make a pass. This method takes the current location of the other six players on the passer's team and computes which player has the greatest space between itself and the corresponding defender. The passer then randomly picks a teammate to receive the pass, but the distance between the receiver and the corresponding defender weights the pick. This is a

great simplification of a real game because there is no concern for which way the marker is forcing or the difficulty of the throw.

3.4 Three-Dimensional Representation

The three-dimensional representation of the ultimate simulator is controlled by four classes: *World3D*, *Field3D*, *Player3D*, and *Disc3D*. *World3D* is the main class that is responsible for rendering everything in the three-dimensional representation. The lighting, camera angles, etc. are all initialized and controlled by the *World3D* class. The *Field3D* class is simply used to set up the background of the three-dimensional window. The field, end zone lines, and marking cones are all created in the *Field3D* class. The *Player3D* and *Disc3D* classes are responsible for the representation of the players and the disc. Figure 7 is an example of one of the blue team players.

Figure 7: Three-Dimensional Player

The players and disc are simply drawn based upon a private *DoublePoint* variable in the class. The players are always drawn based on the x-, y-coordinates of the private variable. The disc is drawn based on the x-, y-, and z-coordinates of the private variable. These private variables are all updated by the respective methods in the *Disc* and *Player* classes. Any method or action that updates the location of the *Disc* and/or *Player* class will also update the location of the *Disc3D* and/or *Player3D* variable.

3.5 Camera Angles

There are four static camera angles available that can be selected from the *CameraControlFrame*: "North End Zone", "South End Zone", "Side Line" and "Overhead." The three-dimensional window allows for manual rotation and sliding around the field, so the user can select any angle or view for the simulation. However, the *CameraControlFrame* is a simple representation that allows the user to bring the camera back to one of the known positions, as shown in Figure 8.

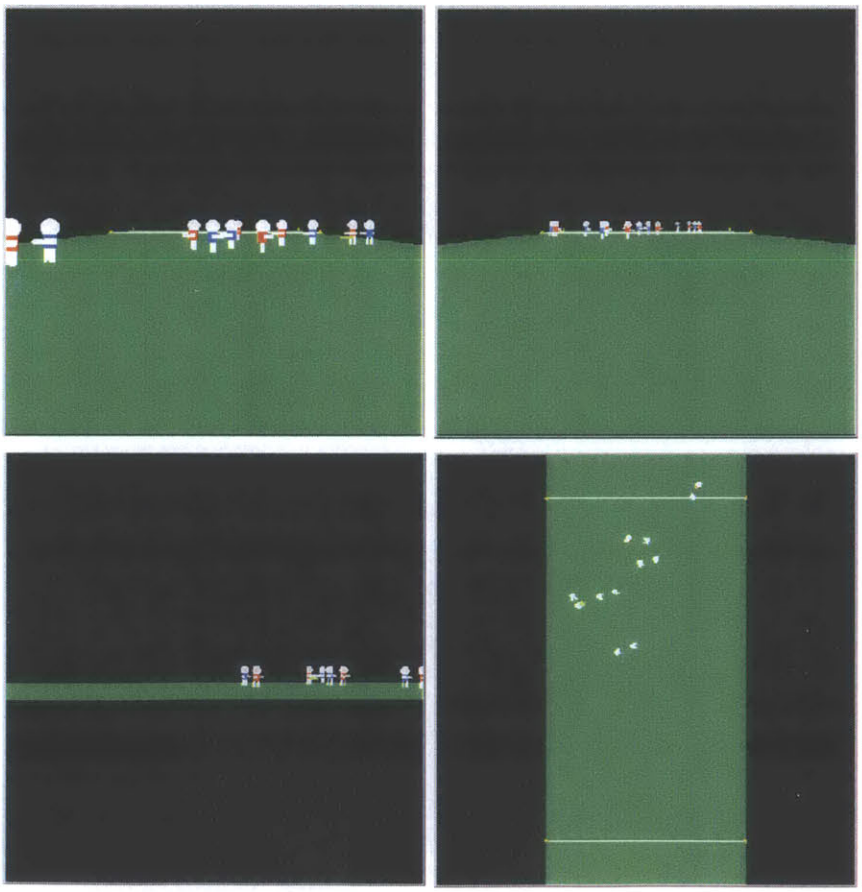

Figure 8: (clockwise from upper left) South End Zone, North End Zone, Side Line and Overhead

The camera viewpoint can be set by the method *setCameraView* in the *World3D* class. The method takes in a *DoublePoint* as well as two *double* parameters, which

correspond to the location of the camera as well as the rotation about the z-axis and the rotation about the x-axis. Figure 9 is a simple diagram of the values for possible rotations about the two axes in radians. For the rotation about the z-axis, 0 radians directs the camera toward the south while –л radians points the camera toward the north. For the rotation around the x-axis, 0 radians directs the camera down toward the field; however, л/2 radians aligns the camera parallel to the playing field.

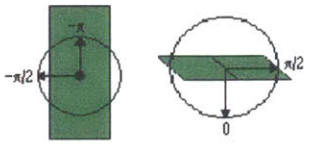

Figure 9: Parameters for z-rotation and x-rotation of the camera (in radians)

3.6 Replay Controls

The replay controller is used to save the state of the simulator. While the simulation is running, the *recordGame* method is called every second. *recordGame* is used to copy the current state of the simulator and store it in the *recordedGames* array list. By copying all of the private variables for the disc and players, the simulation can be paused, rewound, and restarted without additional hassle. This is because the disc stores its own location, velocity and angle. The player stores its own location, its destination, whether it is on offense or defense, as well as any other necessary state information.

The *MAX_RECORDED_GAMES* variable in the *Globals* class specifies how many games are saved in the array list. As the simulation runs, games are added to the array list in a first-in, first-out manner. The simulation control panel as well as the simulation menu allow for navigation of the saved games. The controls operate in a manner similar to the control on a VCR. "Restart" moves the simulation to the earliest saved game. "Rewind" moves the simulation back one second in time. "Play" resumes the simulation from the current saved game. "Pause" stops the simulation. "Fast

forward" advances the simulation ahead one second in time; however, it will only advance to the most recently saved game. "Skip ahead" will advance the simulation to the most recently saved game. Both "fast forward" and "skip ahead" will not have an effect if the simulation has not been previously rewound.

4 Summary

The main purpose of this project was to create a realistic simulation of a game of Ultimate. The simulator should allow a coach to diagram plays and concepts so that a team the fundamentals and intricies of the sport of Ultimate. This simulator lays the framework for such an endeavor. The players are independent with their own aritifcial intelligence. The camera angles can be shown from a variety of static positions. The replay control allows the user to diagram a play multiple times without having to restart the simulator before each reply.

However, there are many more enhancements that can be made to the simulator. For instance, dynamic camera angles would allow for viewing the play through one of the players on the field. Also, enabling the user to reposition players and modify team variables such as the mark would be of great use. Finally, as with any simulation, the artificial intelligence of the players can be augmented to provide more realistic behaviour.

As previously stated, this project provides the basic framework for such augmentation. By implementing a subsumption architecture, the behaviour of the players is easily extendable. Through the creation of multiple classes, the addition of dynamic

camera views becomes simple. The decision to implement the simulator in such a way that future work is not prohibitively difficult was a driving force behind the design of the three-dimensional Ultimate coaching simulator.

References

[UPA 2001] Chris Van Holmes, Troy Frever, Will Deaver, Alan Hoyle, Eric Simon, Joy Endicott, Kate Bergeron, Juha Jalovaara, Tim Murray, Lorne Beckman, and Dan Engstrom. *Ultimate Players Association 10th Edition Rules (10.1).* 2001.

[Brooks 1985] Rodney A. Brooks. *A Robust Layered Control System For A Mobile Robot.* AI Memo. September 1985.

[Menalto 1995] Bharat Mediratta. *Ultimate Playmaker.* http://www.menalto.com/ultimate/about.php. 1995.

Appendix A – Java™ Plug-In Installation Instructions

To run simulation on Windows or Unix:

- The Java™ 2 Platform and Java 3D™ APIs must be installed on your system
  - Go to http://java.sun.com/products/java-media/3D/releases.html
  - If the Java™ 2 Platform is not yet installed, follow the link to install
  - Download the Java 3D™ API file and install
- In order to view the applet, the browser must Java™ Plug-In v1.3.1 or greater
  - For Internet Explorer:
    - Select Tools → Internet Options
    - Go to the Advanced Tab
    - In the Java (Sun) section, "Use Java 2 v1.3.1 for <applet>" must be checked and the version must be 1.3.1 or greater
  - For Netscape:
    - Select Edit → Preferences
    - Go to the Advanced menu
    - "Enable Java" and "Enable Java Plug-In" must be checked
    - Select Help → About Plug-Ins
    - Under the Java Plug-In sections, the "Java Plug-In Version" must be v1.3.1 or higher
- Browse to the simulator by going to http://graphics.lcs.mit.edu/~dbolton
  - Either follow the links to the applet
  - Or download the application from the main page
    - Run "java ultimate.simulator.UltimateApplication" from the command line or shell prompt

To run the simulation of Irix:

- The Java™ 2 Platform must be installed on your system
  - Go to http://java.sun.com/j2se/1.4/download.html
- Install the Java 3D™ Release for SGI Irix
  - Go to http://www.sgi.com/developers/devtools/languages/java3d121.html
  - Follow the download link at the bottom of the page
- Run the simulator via "java ultimate.simulator.UltimateApplication" from a shell prompt

Appendix B – Source Code

AboutFrame.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

//////////////////////////////////////////////////////////////////////
//
//  AboutFrame.java
//
//  Author: Dean Bolton - 2002
//
//    Creates the about frame which pops up from the Help menu
//
//////////////////////////////////////////////////////////////////////


public class AboutFrame extends JFrame implements ActionListener {

    public static final String OKAY = "OK";

    private JPanel contentPane;

    private JPanel aboutPanel;
    private JLabel aboutLabel;

    private JPanel buttonPanel;
    private JButton okayButton;

    public AboutFrame() {
        super("About");

        setDefaultCloseOperation(WindowConstants.HIDE_ON_CLOSE);
        setSize(300, 300);
        contentPane = new JPanel(new BorderLayout());
        contentPane.setBackground(Color.white);
        contentPane.setDoubleBuffered(true);
        setContentPane(contentPane);

        aboutPanel = new JPanel(new FlowLayout());
        aboutPanel.setBackground(Color.white);
        contentPane.add(aboutPanel, "North");

        aboutLabel = new JLabel("3D Ultimate Coaching Simulator");
        aboutLabel.setBackground(Color.black);
        aboutPanel.add(aboutLabel);

        buttonPanel = new JPanel(new FlowLayout());
        buttonPanel.setBackground(Color.white);
        contentPane.add(buttonPanel, "South");

        // Add start button.
        okayButton = new JButton(OKAY);
        okayButton.addActionListener(this);
        buttonPanel.add(okayButton);
    }

    // Implements actionListener to close popup windows
    // after use clicks on the "OK" button
    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == okayButton) {
            this.setVisible(false);
        }
    }
}
```

CameraControlFrame.java

```
package ultimate.simulator;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import java.awt.geom.*;
import java.awt.image.*;

//////////////////////////////////////////////////////////////////////
//
//  CameraControlFrame.java
//
//  Author: Dean Bolton - 2002
//
//    Class for the camera panel which is accessible from the
//    Panels menu.  Implements radio buttons to switch camera
//    angles in the 3D simulation window
//
//////////////////////////////////////////////////////////////////////


public class CameraControlFrame extends JFrame implements ActionListener {

    private World3D world3D = null;

    private JPanel contentPane;

    private JRadioButton camera1;
    private JRadioButton camera2;
    private JRadioButton camera3;
    private JRadioButton camera4;

    public CameraControlFrame(World3D world3D) {
        super("Camera Controls");
        this.world3D = world3D;

        // Set Frame Parameters
        setDefaultCloseOperation(WindowConstants.HIDE_ON_CLOSE);
        setSize(250, 250);

        // Setup Content Pane
        contentPane = new JPanel(new FlowLayout());
        contentPane.setDoubleBuffered(true);
        setContentPane(contentPane);

        // Setup the buttons
        camera1 = new JRadioButton("South End Zone");
        camera1.setActionCommand("Cam1");
        // camera1.setSelected(true);

        camera2 = new JRadioButton("North End Zone");
        camera2.setActionCommand("Cam2");

        camera3 = new JRadioButton("Side Line");
        camera3.setActionCommand("Cam3");

        camera4 = new JRadioButton("Overhead");
        camera4.setActionCommand("Cam4");

        ButtonGroup group = new ButtonGroup();
        group.add(camera1);
        group.add(camera2);
        group.add(camera3);
        group.add(camera4);

        // Register a listener
```

```
        camera1.addActionListener(this);
        camera2.addActionListener(this);
        camera3.addActionListener(this);
        camera4.addActionListener(this);

        contentPane.add(camera1);
        contentPane.add(camera2);
        contentPane.add(camera3);
        contentPane.add(camera4);

    }

    // Implements action listeners for radio buttons on
    // the camera control frame.
    public void actionPerformed(ActionEvent event){
        String act = event.getActionCommand();
        if (act.equals("Cam1")){
            DoublePoint dp =
                new DoublePoint(0,
                                (Globals.BASE_FIELD_LENGTH/2.0 *
                                 Globals.GRAPHICS_3D_MULTIPLIER),
                                2 * Globals.GRAPHICS_3D_MULTIPLIER);

            // setCameraView changes the location of the camera
            // in the 3D graphics panel.  The first parameter is
            // a location DoublePoint that sets the location of
            // the camera in the three-dimensional coordinate
            // system with (0, 0, 0) being the center of the
            // playing field.  The second parameter is in radians
            // and specifies the z-rotation of the camera.  0
            // points toward the south of the field, and -PI
            // points toward the north.  The third parameter
            // is also in radians and specifies the x-rotation
            // of the camera.  PI/2 points the camera parallel
            // to the field.  0 points the camera directly
            // down toward the playing field.
            world3D.setCameraView(dp, -Math.PI, Math.PI/2);
        }
        else if (act.equals("Cam2")){
            DoublePoint dp =
                new DoublePoint(0, -(Globals.BASE_FIELD_LENGTH/2.0 *
                                     Globals.GRAPHICS_3D_MULTIPLIER),
                                2 * Globals.GRAPHICS_3D_MULTIPLIER);
            world3D.setCameraView(dp, 0, Math.PI/2);
        }
        else if (act.equals("Cam3")){
            DoublePoint dp =
                new DoublePoint((Globals.BASE_FIELD_WIDTH/2 + 30) *
                                Globals.GRAPHICS_3D_MULTIPLIER, 0,
                                2 * Globals.GRAPHICS_3D_MULTIPLIER);
            world3D.setCameraView(dp, Math.PI/2, Math.PI/2);
        }
        else if (act.equals("Cam4")){
            DoublePoint dp =
                new DoublePoint(0, 0,
                                90 * Globals.GRAPHICS_3D_MULTIPLIER);
            world3D.setCameraView(dp, 0, 0);
        }

    }

}
```

ControlPanel.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.net.*;

//////////////////////////////////////////////////////////////////////
//
//  ControlPanel.java
//
//  Author: Dean Bolton - 2002
//
//    Implements the JPanel which displays the score, the stall count,
//    the player with possession of the disc, the team that is on
//    offense, and other relevant information about the game
//
//////////////////////////////////////////////////////////////////////


public class ControlPanel extends JPanel {

    private Ultimate uf;
    private JLabel redScore;
    private JLabel blueScore;
    private JLabel redName;
    private JLabel blueName;
    private ImageIcon discIcon;
    private ImageIcon blankIcon;
    private JLabel stallCountTime;
    private JLabel possessorName;
    private JLabel possessorPlayer;

    public ControlPanel(Ultimate uf, URL codeBase) {

        super(new BorderLayout(10, 10));
        this.uf = uf;

        Border controlBorder = new CompoundBorder(new EtchedBorder(),
                                                  new EmptyBorder(5,5,5,5));
        setBorder(controlBorder);

        JPanel upperPanel = new JPanel(new BorderLayout());
        add(upperPanel, "North");

        JPanel scoreBoard = new JPanel(new GridLayout(2, 2, 5, 5));
        scoreBoard.setBorder(controlBorder);
        upperPanel.add(scoreBoard, "North");

        if (codeBase != null) {

            try {

                discIcon = new ImageIcon(new URL(codeBase,
                                                 "images/disc.gif"));
                blankIcon = new ImageIcon(new URL(codeBase,
                                                  "images/blank.gif"));
            } catch (MalformedURLException e) {

                System.out.println(e.toString());
            }
        } else {

            discIcon = new ImageIcon("images/disc.gif");
            blankIcon = new ImageIcon("images/blank.gif");
        }
```

```
    // Initializes the score board.
    Font scoreFont = new Font("Dialog", Font.BOLD, 15);
    redName = new JLabel("Red: ", blankIcon, JLabel.RIGHT);
    redName.setFont(scoreFont);
    scoreBoard.add(redName);

    redScore = new JLabel("00", JLabel.CENTER);
    redScore.setFont(scoreFont);
    scoreBoard.add(redScore);

    blueName = new JLabel("Blue: ", blankIcon, JLabel.RIGHT);
    blueName.setFont(scoreFont);
    scoreBoard.add(blueName);

    blueScore = new JLabel("00", JLabel.CENTER);
    blueScore.setFont(scoreFont);
    scoreBoard.add(blueScore);

    JPanel stallCountPanel = new JPanel(new BorderLayout());
    stallCountPanel.setBorder(controlBorder);
    upperPanel.add(stallCountPanel, "Center");

    JLabel stallCountName = new JLabel("Stall Count", JLabel.CENTER);
    stallCountName.setFont(scoreFont);
    stallCountPanel.add(stallCountName, "North");

    stallCountTime = new JLabel("10", JLabel.CENTER);
    stallCountTime.setFont(new Font("Dialog", Font.BOLD, 20));
    stallCountPanel.add(stallCountTime, "Center");

    JPanel possessorPanel = new JPanel(new GridLayout(2, 1));
    possessorPanel.setBorder(controlBorder);
    upperPanel.add(possessorPanel, "South");

    possessorName = new JLabel("Possessor: ", JLabel.CENTER);
    possessorName.setFont(scoreFont);
    possessorPanel.add(possessorName);

    possessorPlayer = new JLabel("", JLabel.CENTER);
    possessorPlayer.setFont(scoreFont);
    possessorPanel.add(possessorPlayer);
}

// updates the score board after each point is scored
public void updateScores() {

    int blueScoreValue = uf.getGame().getBlueTeam().getScore();
    String blueScoreString = "" + blueScoreValue;
    if (blueScoreValue < 10)
        blueScoreString = "0" + blueScoreValue;
    blueScore.setText(blueScoreString);

    int redScoreValue = uf.getGame().getRedTeam().getScore();
    String redScoreString = "" + redScoreValue;
    if (redScoreValue < 10)
        redScoreString = "0" + redScoreValue;
    redScore.setText(redScoreString);
}

// updates the team with possession of the disc
public void updatePossession() {

    if (uf.getGame().getRedTeam().hasPossession())
        redName.setIcon(discIcon);
    else
        redName.setIcon(blankIcon);

    if (uf.getGame().getBlueTeam().hasPossession())
        blueName.setIcon(discIcon);
    else
        blueName.setIcon(blankIcon);
```

```
    }

    // updates the stall count
    public void updateStallCount() {

        int stallCountValue = (int) Math.round(uf.getGame().getStallCount());
        String stallCountString = "" + stallCountValue;
        if (stallCountValue < 10)
            stallCountString = "0" + stallCountValue;
        stallCountTime.setText(stallCountString);
    }

    // updates the player with possession of the disc
    public void updatePossessingPlayer() {

        Player player = uf.getGame().getRedTeam().getPossessingPlayer();
        if (player != null) {

            possessorPlayer.setText(player.getName());
            possessorPlayer.setForeground(Color.red);
        } else {
            player = uf.getGame().getBlueTeam().getPossessingPlayer();
            if (player != null) {

                possessorPlayer.setText(player.getName());
                possessorPlayer.setForeground(Color.blue);
            } else {

                possessorPlayer.setText("");
            }
        }
    }

    // changes the display after one team wins the game
    public void displayWinningTeam() {

        possessorName.setText("Winner: ");

        if (uf.getGame().getRedTeam().getScore() ==
            Globals.GAME_WINNING_POINTS) {

            possessorPlayer.setText("Red has won!");
            possessorPlayer.setForeground(Color.red);
        } else {

            possessorPlayer.setText("Blue has won!");
            possessorPlayer.setForeground(Color.blue);
        }
    }
}
```

Disc.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.media.j3d.*;

//////////////////////////////////////////////////////////////////////////
//
//  Disc.java
//
//  Author: Dean Bolton - 2002
//
//    Extension of DoublePoint.java which adds additonal
//    functionality such as the private disc3D, velocity,
//    and angle variables
//
//////////////////////////////////////////////////////////////////////////


public class Disc extends DoublePoint {

    public static final double DROP_RATE = .4D;
    public static final double THROWING_HEIGHT = 1D;

    private double velocity = 0D;
    private double angle = 0D;
    private Disc3D disc3D = null;

    public Disc() {
        super(0D, 0D, 0D);

        double xLocation3D =
            this.getXLocation() / Globals.GRAPHICS_3D_CONVERSION;
        double yLocation3D =
            this.getYLocation() / Globals.GRAPHICS_3D_CONVERSION;
        double zLocation3D =
            this.getZLocation() / Globals.GRAPHICS_3D_CONVERSION;

        disc3D = new Disc3D(new DoublePoint(xLocation3D,
                                            yLocation3D,
                                            zLocation3D));
    }

    public void fly(double velocity, double angle) {

        this.velocity = velocity;
        this.angle = angle;
    }

    // updates the position of the disc based on
    // the velocity, angle and time parameter
    public void action(double time) {

        if (velocity > 0D) {

            double distanceCovered = velocity * time;
            super.moveTo(angle, distanceCovered);
            super.setZLocation(super.getZLocation() - (DROP_RATE * time));

            if (super.getZLocation() <= 0D) {

                super.setZLocation(0D);
                velocity = 0D;
            }

            // updates the position of the disc3D
            double xLocation3D =
```

```
            this.getXLocation() / Globals.GRAPHICS_3D_CONVERSION;
        double yLocation3D =
            this.getYLocation() / Globals.GRAPHICS_3D_CONVERSION;
        double zLocation3D =
            this.getZLocation() / Globals.GRAPHICS_3D_CONVERSION;

        disc3D.setLocation(new DoublePoint(xLocation3D,
                                           yLocation3D,
                                           zLocation3D));
    }
}

public double getVelocity() {

    return velocity;
}

public void setVelocity(double velocity) {

    this.velocity = velocity;
}

// updates the position of the disc3D
public void update() {

    double xLocation3D =
        this.getXLocation() / Globals.GRAPHICS_3D_CONVERSION;
    double yLocation3D =
        this.getYLocation() / Globals.GRAPHICS_3D_CONVERSION;
    double zLocation3D =
        this.getZLocation() / Globals.GRAPHICS_3D_CONVERSION;

    disc3D.setLocation(new DoublePoint(xLocation3D,
                                       yLocation3D,
                                       zLocation3D));
}

public TransformGroup getTransformGroup() {

    return disc3D.getTransformGroup();
}

// used by the Replay Controller in order to save
// the game each second
public Disc copyDisc() {

    Disc cloneDisc = new Disc();
    cloneDisc.setToLocation(this.getLocation());
    cloneDisc.velocity = velocity;
    cloneDisc.angle = angle;
    cloneDisc.disc3D = disc3D;

    return cloneDisc;
}
}
```

Disc3D.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.geometry.*;
import javax.vecmath.*;

////////////////////////////////////////////////////////////////////////
//
//  Disc3D.java
//
//  Author: Dean Bolton - 2002
//
//    Class that stores all of the information pertaining to
//    the three-dimension representation of the disc
//
////////////////////////////////////////////////////////////////////////


public class Disc3D {

    private DoublePoint location = null;

    private TransformGroup group = null;
    private TransformGroup outerGroup = null;
    private Transform3D rotationTransform = null;
    private Transform3D translateTransform = null;

    public Disc3D(DoublePoint location) {

        this.location = location;

        rotationTransform = new Transform3D();
        translateTransform = new Transform3D();
        outerGroup = new TransformGroup();
        group = new TransformGroup();

        outerGroup.addChild(group);

        outerGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
         outerGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

         group.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
         group.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    }

    public DoublePoint getLocation() {

        return location;
    }

    public void setLocation(DoublePoint location) {

        this.location = location;
        translateTransform.set(new Vector3d(location.getXLocation(),
                                            location.getYLocation(),
                                            location.getZLocation()));
        updateTransform();
    }

    private void updateTransform() {

        outerGroup.setTransform(translateTransform);
        group.setTransform(rotationTransform);
    }
```

```
    public TransformGroup getTransformGroup() {

        Appearance appearance = new Appearance();
        ColoringAttributes coloringAttributes =
            new ColoringAttributes(new Color3f(Color.yellow),
                                   ColoringAttributes.FASTEST);
        appearance.setColoringAttributes(coloringAttributes);

        Cylinder discShape = new Cylinder(.08f, .03f);
        discShape.setAppearance(appearance);

        Transform3D rotate = new Transform3D();
        rotate.rotX(Math.PI / 2D);
        TransformGroup objectRotate = new TransformGroup(rotate);

        group.addChild(objectRotate);
        objectRotate.addChild(discShape);

        return outerGroup;
    }
}
```

DoublePoint.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

//////////////////////////////////////////////////////////////////////
//
//  DoublePoint.java
//
//  Author: Dean Bolton - 2002
//
//    Basic class that stores a three-dimensional coordinate
//    along with methods for computing the angle and distance
//    from the current location to another location
//
//////////////////////////////////////////////////////////////////////


public class DoublePoint {

    private double xLocation;
    private double yLocation;
    private double zLocation;

    public DoublePoint(double xLocation, double yLocation, double zLocation) {

        this.xLocation = xLocation;
        this.yLocation = yLocation;
        this.zLocation = zLocation;
    }

    public DoublePoint(double xLocation, double yLocation) {

        this(xLocation, yLocation, 0D);
    }

    public DoublePoint(DoublePoint startLocation) {

        this(0D, 0D, 0D);
        setToLocation(startLocation);
    }

    public double getXLocation() {

        return xLocation;
    }

    public void setXLocation(double xLocation) {

        this.xLocation = xLocation;
    }
    public double getYLocation() {

        return yLocation;
    }

    public void setYLocation(double yLocation) {

        this.yLocation = yLocation;
    }
    public double getZLocation() {

        return zLocation;
    }

    public void setZLocation(double zLocation) {
```

```
        this.zLocation = zLocation;
    }

    public void setToLocation(DoublePoint destination) {

        xLocation = destination.getXLocation();
        yLocation = destination.getYLocation();
        zLocation = destination.getZLocation();
    }

    public DoublePoint getLocation() {

        return (new DoublePoint(xLocation, yLocation, zLocation));
    }

    // returns the distance from the current location to
    // the destination parameter
    public double getDistanceToPoint(DoublePoint destination) {

        double diffX = (destination.getXLocation() - xLocation);
        double diffY = (destination.getYLocation() - yLocation);
        double diffZ = (destination.getZLocation() - zLocation);

        return Math.sqrt((diffX * diffX) + (diffY * diffY) + (diffZ * diffZ));
    }

    // returns true if the current location is equal to
    // the destination parameter; otherwise, returns false
    public boolean reachedPoint(DoublePoint destination) {

        boolean xReached = (Math.abs(xLocation - destination.getXLocation()) < .001);
        boolean yReached = (Math.abs(yLocation - destination.getYLocation()) < .001);
        boolean zReached = (Math.abs(zLocation - destination.getZLocation()) < .001);

        return (xReached && yReached && zReached);
    }

    // returns the angle from the current location to
    // the destination parameter based on their x,y-coordinates
    public double getAngleToPoint(DoublePoint destination) {

        double diffX = (destination.getXLocation() - xLocation);
        double diffY = (destination.getYLocation() - yLocation);

        if (diffX == 0D)
            diffX += .00001D;

        double angle = Math.atan(diffY / diffX);

        if (diffX < 0D)
            angle -= Math.PI;

        if (angle < 0D)
            angle = (2D * Math.PI) + angle;
        else if (angle > (2D * Math.PI))
            angle -= (2D * Math.PI);

        return angle;
    }

    // updates the current location based on the distance
    // and angle input parameters
    public void moveTo(double angle, double distance) {

        xLocation += distance * Math.cos(angle);
        yLocation += distance * Math.sin(angle);
    }
}
```

Field3D.java

```
package ultimate.simulator;

import java.awt.*;
import com.sun.j3d.utils.applet.*;
import java.awt.event.*;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.*;
import javax.media.j3d.*;
import javax.vecmath.*;

////////////////////////////////////////////////////////////////////////
//
//  Field3D.java
//
//  Author: Dean Bolton - 2002
//
//    Class for the three-dimensional representation of the field
//
////////////////////////////////////////////////////////////////////////


public class Field3D{

    private TransformGroup fieldGraph = null;

    public Field3D() {

        fieldGraph = new TransformGroup();
        fieldGraph.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        fieldGraph.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

       // sets up the field itself
       QuadArray field =
           new QuadArray(4, QuadArray.COORDINATES | QuadArray.COLOR_3);

       double fieldWidth = (Globals.BASE_FIELD_WIDTH /
                            Globals.GRAPHICS_3D_CONVERSION);
       double fieldLength = (Globals.BASE_FIELD_LENGTH /
                           Globals.GRAPHICS_3D_CONVERSION);
       double fieldBoundLeftX = -(fieldWidth / 2);
       double fieldBoundRightX = (fieldWidth / 2);
       double fieldBoundTopY = -(fieldLength / 2);
       double fieldBoundBottomY = (fieldLength / 2);
       double endZoneLength = (Globals.BASE_ENDZONE_LENGTH /
                               Globals.GRAPHICS_3D_CONVERSION);

        field.setCoordinate(0, new Point3d(fieldBoundLeftX, fieldBoundTopY, 0D));
        field.setCoordinate(1, new Point3d(fieldBoundRightX, fieldBoundTopY, 0D));
        field.setCoordinate(2, new Point3d(fieldBoundRightX, fieldBoundBottomY, 0D));
        field.setCoordinate(3, new Point3d(fieldBoundLeftX, fieldBoundBottomY, 0D));

        field.setColor(0, new Color3f(Color.green));
        field.setColor(1, new Color3f(Color.green));
        field.setColor(2, new Color3f(Color.green));
        field.setColor(3, new Color3f(Color.green));

       // sets up the white goal lines
       LineArray redGoalLine =
           new LineArray(2, LineArray.COORDINATES | LineArray.COLOR_3);
       LineArray blueGoalLine =
           new LineArray(2, LineArray.COORDINATES | LineArray.COLOR_3);

       redGoalLine.setCoordinate(0, new Point3d(fieldBoundLeftX,
                                          fieldBoundTopY + endZoneLength,
                                          0.01D));
       redGoalLine.setCoordinate(1, new Point3d(fieldBoundRightX,
                                          fieldBoundTopY + endZoneLength,
                                          0.01D));
```

```
        blueGoalLine.setCoordinate(0, new Point3d(fieldBoundLeftX,
                                                  fieldBoundBottomY - endZoneLength,
                                                  0.01D));
        blueGoalLine.setCoordinate(1, new Point3d(fieldBoundRightX,
                                                  fieldBoundBottomY - endZoneLength,
                                                  0.01D));

        redGoalLine.setColor(0, new Color3f(Color.white));
        redGoalLine.setColor(1, new Color3f(Color.white));
        blueGoalLine.setColor(0, new Color3f(Color.white));
        blueGoalLine.setColor(1, new Color3f(Color.white));

         fieldGraph.addChild(new Shape3D(field));
        fieldGraph.addChild(new Shape3D(redGoalLine));
        fieldGraph.addChild(new Shape3D(blueGoalLine));

        // adds the eight orange cones
        Appearance appearance = new Appearance();
        ColoringAttributes coloringAttributes =
            new ColoringAttributes(new Color3f(Color.orange),
                                   ColoringAttributes.FASTEST);
        appearance.setColoringAttributes(coloringAttributes);

        addCone(fieldBoundLeftX, fieldBoundTopY, appearance);
        addCone(fieldBoundLeftX, fieldBoundTopY + endZoneLength, appearance);
        addCone(fieldBoundLeftX, fieldBoundBottomY - endZoneLength, appearance);
        addCone(fieldBoundLeftX, fieldBoundBottomY, appearance);
        addCone(fieldBoundRightX, fieldBoundTopY, appearance);
        addCone(fieldBoundRightX, fieldBoundTopY + endZoneLength, appearance);
        addCone(fieldBoundRightX, fieldBoundBottomY - endZoneLength, appearance);
        addCone(fieldBoundRightX, fieldBoundBottomY, appearance);
    }

    public TransformGroup getFieldGraph() {

        return fieldGraph;
    }

    // method for adding cones to the field transform group
    public void addCone(double xLocation, double yLocation, Appearance appearance) {

        Cone cone = new Cone(0.05f, 0.1f);
        cone.setAppearance(appearance);

        Transform3D rotateTransform = new Transform3D();
        Transform3D coneTransform = new Transform3D();
        rotateTransform.rotX(Math.PI/2.0D);
        coneTransform.set(new Vector3d(xLocation, yLocation, 0.05D));
        TransformGroup rotationTransformGroup = new TransformGroup(rotateTransform);
        TransformGroup coneTransformGroup = new TransformGroup(coneTransform);
        rotationTransformGroup.addChild(cone);
        coneTransformGroup.addChild(rotationTransformGroup);

        fieldGraph.addChild(coneTransformGroup);
    }
}
```

FieldControlFrame.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.net.*;

//////////////////////////////////////////////////////////////////////
//
//  FieldControlFrame.java
//
//  Author: Dean Bolton - 2002
//
//    Basic popup frame for the two-dimensional playing
//    field and the control panel
//
//////////////////////////////////////////////////////////////////////


public class FieldControlFrame extends JFrame {

    private ControlPanel control = null;

    public FieldControlFrame(Ultimate uf, boolean isApplet,
                             URL codeBase, Game game) {

        super("Field Controls");

        setDefaultCloseOperation(WindowConstants.HIDE_ON_CLOSE);
        setSize(500, 700);
        setLocation(500, 0);
        JPanel contentPane = new JPanel(new BorderLayout(10, 10));
        contentPane.setDoubleBuffered(true);
        this.setContentPane(contentPane);

        PlayingField field = new PlayingField(uf, isApplet);
        this.getContentPane().add(field, "Center");

        control = new ControlPanel(uf, codeBase);
        this.getContentPane().add(control, "East");
        game.setControlPanel(control);
    }
}
```

Game.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

//////////////////////////////////////////////////////////////////////
//
//  Game.java
//
//  Author: Dean Bolton - 2002
//
//    Class that stores all of the information pertaining to
//    the game.  Initializes the two teams and controls the
//    state of the game.
//
//////////////////////////////////////////////////////////////////////


public class Game {

    // simple ints used as the status of the game
    public static final int ERROR = 0;
    public static final int PRE_PULL = 1;
    public static final int PULL = 2;
    public static final int PLAY = 3;
    public static final int FOUL = 4;
    public static final int OUT_OF_BOUNDS = 5;
    public static final int PICK = 6;
    public static final int SCORE = 7;
    public static final int END_OF_GAME = 8;

    private int status;
    private ControlPanel controlPanel;
    private double stallCount = Globals.STALL_COUNT;
    private boolean runGameFlag;
    private boolean stallCountRunning = false;
    private Team redTeam = null;
    private Team blueTeam = null;
    private Disc disc = null;
    private Player possessingPlayer = null;
    private Team pullTeam = null;
    private boolean discFetch = false;

    public Game(boolean populate) {

        if (populate) {

            // initializes the disc and teams at the start
            // of the simulator
            disc = new Disc();
            redTeam = new Team(Team.RED_TEAM, "Red", false, true, this);
            blueTeam = new Team(Team.BLUE_TEAM, "Blue", true, false, this);
            redTeam.setMarkedPlayers();
            blueTeam.setMarkedPlayers();

            runGameFlag = false;
            status = PRE_PULL;
        }
    }

    // game loop that controls the flow of the game
    public void runGame(double time) {

        if (runGameFlag) {

            if (status == PRE_PULL) {
                // used to set up the teams before the pull
```

```
            // player3 is always selected to pull the disc

            if (atPullPositions()) {

                if (redTeam.hasPossession()) {

                    possessingPlayer = redTeam.getPlayer(3);
                    pullTeam = redTeam;
                } else {

                    possessingPlayer = blueTeam.getPlayer(3);
                    pullTeam = blueTeam;
                }

                possessingPlayer.setPossessDisc(true);
                controlPanel.updatePossession();
                status = PULL;
            } else
                goToPullPositions();
        } else if (status == PULL) {
            // sets up the actual pull

            Team possessingTeam = null;
            if (redTeam.hasPossession())
                possessingTeam = redTeam;
            else
                possessingTeam = blueTeam;

            if (possessingTeam != pullTeam) {

                status = PLAY;
            } else {

                redTeam.pull(blueTeam.hasPossession());
                blueTeam.pull(redTeam.hasPossession());
            }
        } else if (status == END_OF_GAME) {
            // status after one team has one the game

            controlPanel.displayWinningTeam();
            return;
        }

        if (stallCountRunning) {

            stallCount -= time;
            controlPanel.updateStallCount();
        }

        // updates the control panel and moves
        // the disc and players
        controlPanel.updatePossessingPlayer();
        disc.action(time);
        movePlayers(time);
    }
}

public void startStallCount() {

    stallCountRunning = true;
    stallCount = Globals.STALL_COUNT;
}

public void stopStallCount() {

    stallCountRunning = false;
    stallCount = Globals.STALL_COUNT;
    controlPanel.updateStallCount();
}

private boolean atPullPositions() {
```

```
    if (redTeam.atPullPositions() &&
        blueTeam.atPullPositions())
        return true;
    else
        return false;
}

private void goToPullPositions() {

    redTeam.goToPullPositions();
    blueTeam.goToPullPositions();
}

private void movePlayers(double time) {

    for (int x=0; x<7; x++) {

        redTeam.getPlayer(x).action(time);
        blueTeam.getPlayer(x).action(time);
    }
}

public void changePossession() {

    redTeam.changePossession();
    blueTeam.changePossession();
    controlPanel.updatePossession();
}

public double getStallCount() {

    return stallCount;
}

public Disc getDisc() {

    return disc;
}

public Team getRedTeam() {

    return redTeam;
}

public Team getBlueTeam() {

    return blueTeam;
}

public void runGame() {

    runGameFlag = true;
}

public void pauseGame() {

    runGameFlag = false;
}

public boolean isRunning() {

    return runGameFlag;
}

public void goalScored() {

    controlPanel.updateScores();
    redTeam.clearPlayerPossession();
    blueTeam.clearPlayerPossession();
```

```
        redTeam.setPlayingFieldSide(!redTeam.getPlayingFieldSide());
        blueTeam.setPlayingFieldSide(!blueTeam.getPlayingFieldSide());

        redTeam.determinePullPositions();
        blueTeam.determinePullPositions();

        if (redTeam.getScore() == Globals.GAME_WINNING_POINTS ||
            blueTeam.getScore() == Globals.GAME_WINNING_POINTS) {

            status = END_OF_GAME;
        } else {

            status = PRE_PULL;
        }
    }

    public void setControlPanel(ControlPanel cPane) {

        this.controlPanel = cPane;
    }

    public void refreshControlPanel() {

        if (controlPanel != null) {

            controlPanel.updateScores();
            controlPanel.updatePossession();
            controlPanel.updateStallCount();
            controlPanel.updatePossessingPlayer();
        }
    }

    // used by the Replay Controller to copy the game
    public Game copyGame() {

        Game cloneGame = new Game(false);
        cloneGame.disc = disc.copyDisc();
        cloneGame.redTeam = redTeam.copyTeam(cloneGame);
        cloneGame.blueTeam = blueTeam.copyTeam(cloneGame);
        cloneGame.redTeam.setMarkedPlayers();
        cloneGame.blueTeam.setMarkedPlayers();
        cloneGame.status = status;
        cloneGame.controlPanel = controlPanel;
        cloneGame.stallCount = stallCount;
        cloneGame.runGameFlag = runGameFlag;
        cloneGame.stallCountRunning = stallCountRunning;
        cloneGame.discFetch = discFetch;

        if (pullTeam == redTeam)
            cloneGame.pullTeam = cloneGame.redTeam;
        else
            cloneGame.pullTeam = cloneGame.blueTeam;

        cloneGame.possessingPlayer = redTeam.getPossessingPlayer();
        cloneGame.possessingPlayer = blueTeam.getPossessingPlayer();

        return cloneGame;
    }
}
```

Globals.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

//////////////////////////////////////////////////////////////////////
//
//  Globals.java
//
//  Author: Dean Bolton - 2002
//
//    Stores all of the pertinent global variables
//
//////////////////////////////////////////////////////////////////////


public class Globals {

    public static final int GAME_WINNING_POINTS = 15;
    public static final double STALL_COUNT = 10D;

    public static final double BASE_FIELD_WIDTH = 37D;
    public static final double BASE_FIELD_LENGTH = 110D;
    public static final double BASE_PLAYING_LENGTH = 64D;
    public static final double BASE_ENDZONE_LENGTH = 23D;

    public static final int MAX_RECORDED_GAMES = 30;

    public static final double GRAPHICS_3D_CONVERSION = 7D;
    public static final double GRAPHICS_3D_MULTIPLIER = 0.1482D;
}
```

Player.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.media.j3d.*;

//////////////////////////////////////////////////////////////////////
//
//  Player.java
//
//  Author: Dean Bolton - 2002
//
//    Player class controls the state of the player, the
//    location, and the movement
//
//////////////////////////////////////////////////////////////////////


public class Player {

    // group of ints used to control the state of the player
    public static final int STOP = 0;
    public static final int MOVE_TO = 1;
    public static final int DEFEND = 2;
    public static final int OFFENSE = 3;
    public static final int THROW = 4;
    public static final int POST_THROW_OFFENSE = 5;
    public static final int PULL = 6;
    public static final int CHASE_DISC = 7;

    // max time (in sec) paused after making a cut
    public static final double DESTINATION PAUSE = 5D;

    // simple percentages for catching a disc
    private static final double OFFENSE_CATCH_CHANCE = .3D;
    private static final double DEFENSE_CATCH_CHANCE = .1D;

    // variables used to make defensive adjustments
    private static final double DEFENSIVE_ADJUSTMENT_DISTANCE = 4D;
    private static final double DEFENSIVE_ADJUSTMENT_FACTOR = .9D;
    private static final double DISC_PURSUE_DISTANCE = 20D;
    private static final double DEFENSIVE_COVER_BUFFER_DISTANCE = .4D;
    private static final double BASE_SPEED = 7D;

    private double speed = 0D;
    private double jump = 0D;
    private double strength = 0D;
    private double dexterity = 0D;
    private double stamina = 0D;

    private int command = 0;
    private double destinationTimer = 0D;
    private double stallCount = 0D;
    private boolean stallCountRun = false;
    private int id;
    private String name = "";
    private boolean possessDisc = false;
    private Player markedPlayer = null;
    private Team team = null;
    private DoublePoint location = null;
    private DoublePoint destination = null;
    private Player3D player3D = null;

    public Player(double speed, double jump, double strength,
                  double dexterity, double stamina, int id,
                  String name, Team team, DoublePoint location) {
```

```
    this.speed = speed;
    this.jump = jump;
    this.strength = strength;
    this.dexterity = dexterity;
    this.stamina = stamina;
    this.id = id;
    this.name = name;
    this.team = team;
    this.location = location;
    destination = new DoublePoint(0D, 0D, 0D);

    // sets the location of the 3D player
    double xLocation3D = (location.getXLocation() /
                          Globals.GRAPHICS_3D_CONVERSION);
    double yLocation3D = (location.getYLocation() /
                          Globals.GRAPHICS_3D_CONVERSION);
    double zLocation3D = (location.getZLocation() /
                          Globals.GRAPHICS_3D_CONVERSION);
    double angle = location.getAngleToPoint(destination);

    player3D = new Player3D(new DoublePoint(xLocation3D,
                                            yLocation3D,
                                            zLocation3D),
                            angle,
                            team.getId());

    possessDisc = false;
    command = STOP;
}

public void moveTo(DoublePoint destination) {

    this.destination = destination;
    if (!location.reachedPoint(destination))
        command = MOVE_TO;
}

// basic defend method that has the player follow around
// the offensive player with lags to represent reaction times
public void defend(double time) {

    destination = new DoublePoint(markedPlayer.getLocation());
    double dist = location.getDistanceToPoint(destination) -
        DEFENSIVE_COVER_BUFFER_DISTANCE;
    if (dist > 2D) {

        if (dist < DEFENSIVE_ADJUSTMENT_DISTANCE)
            movePlayer(time * DEFENSIVE_ADJUSTMENT_FACTOR);
        else
            movePlayer(time);
    }

    if (catchDisc(false))
        team.setPossession(true);
}

// simple offensive strategy.  players choose a new spot
// on the field between the disc and the back of the end
// zone.  then they run to that point before selecting a
// location to cut toward.
public void offense(Disc disc, double time) {

    if ((disc.getVelocity() > 0D) &&
        (location.getDistanceToPoint(disc.getLocation()) < DISC_PURSUE_DISTANCE)) {

        destination = new DoublePoint(disc.getLocation());
        destinationTimer = 0D;
    } else {

        destinationTimer -= time;
        if (destinationTimer <= 0D) {
```

```
            destinationTimer = DESTINATION_PAUSE * Math.random();

            double xDestination = (37D * Math.random()) - 18.5D;
            double yDestination = 0D;

            if  (!team.getPlayingFieldSide()) {

                yDestination = ((disc.getYLocation() + 50D) *
                               Math.random()) - 50D;
            } else {

                yDestination = ((50D - disc.getYLocation()) *
                               Math.random()) + disc.getYLocation();
            }
            destination = new DoublePoint(xDestination, yDestination);
        }
    }
    if (!location.reachedPoint(destination))
        movePlayer(time);
    catchDisc(true);
}

// selects a random time before the stall count runs
// down and then throws to a teammate based on
// Team.findOpenReceiver() method at that time
public void throwing(double time) {

    if (!stallCountRun) {
        stallCountRun = true;
        stallCount = Globals.STALL_COUNT * Math.random();
    }

    stallCount -= time;
    if (stallCount <= 0D) {

        Player receiver = team.findOpenReceiver(this);
        throwDisc(receiver.getLocation());
        command = POST_THROW_OFFENSE;
        destinationTimer = 0D;
        stallCountRun = false;
    }
}

public void action(double time) {

    Disc disc = team.getGame().getDisc();
    if (command == STOP) {

    } else if (command == MOVE_TO) {

        if (location.reachedPoint(destination))
            command = STOP;
        else
            movePlayer(time);
    } else if (command == DEFEND) {

        defend(time);
    } else if (command == OFFENSE) {

        offense(disc, time);
    } else if (command == POST_THROW_OFFENSE) {

        if (destinationTimer == 0D) {

            destinationTimer++;
            double xDestination = (37D * Math.random()) - 18.5D;
            double yDestination = 0D;

            if (!team.getPlayingFieldSide()) {
```

```
                yDestination = ((disc.getYLocation() + 50D) *
                               Math.random()) - 50D;
            } else {

                yDestination = ((50D - disc.getYLocation()) *
                               Math.random()) + disc.getYLocation();
            }
            destination = new DoublePoint(xDestination, yDestination);
        }
        if (disc.getVelocity() == 0D)
            command = OFFENSE;
        else
            movePlayer(time);
    } else if (command == THROW) {

        throwing(time);
    } else if (command == PULL) {

        boolean side = team.getPlayingFieldSide();
        double xDestination = ((Globals.BASE_FIELD_WIDTH * Math.random()) -
                              (Globals.BASE_FIELD_WIDTH / 2D));
        double yDestination = 0D;
        if (!side)
            yDestination = 0D - Globals.BASE_FIELD_LENGTH * Math.random() / 2D;
        else
            yDestination = Globals.BASE_FIELD_LENGTH * Math.random() / 2D;

        DoublePoint destination = new DoublePoint(xDestination, yDestination);
        throwDisc(destination);
    } else if (command == CHASE_DISC) {

        destination = new DoublePoint(disc.getLocation());

        if (location.reachedPoint(destination)) {

            setPossessDisc(true);
            if (!team.hasPossession())
                team.getGame().changePossession();
            team.getGame().startStallCount();
            command = THROW;
        } else {

            movePlayer(time);
        }
        catchDisc(true);
    }
}

// if the player is close to the disc, it will attempt
// to catch the disc and update the game if it is successful
private boolean catchDisc(boolean offense) {

    Disc disc = team.getGame().getDisc();
    if (!possessDisc) {

        if (disc.getZLocation() > 0D) {

            if (location.getDistanceToPoint(disc.getLocation()) < 1D) {

                double r = Math.random();
                boolean caught = false;
                if (offense && r <= OFFENSE_CATCH_CHANCE)
                    caught = true;
                else if (!offense && r < DEFENSE_CATCH_CHANCE)
                    caught = true;

                if (caught) {

                    if (PlayingField.isEndZone(!team.getPlayingFieldSide(),
                                               location)) {
```

```
                    disc.setVelocity(0D);
                    disc.setToLocation(location);
                    disc.setZLocation(0D);
                    team.incrementScore();
                    team.getGame().goalScored();
                } else {

                    setPossessDisc(true);
                    if (!team.hasPossession())
                        team.getGame().changePossession();
                    team.getGame().startStallCount();
                    command = THROW;
                }
                return true;
            }
        }
    }
    return false;
}

// updates the player and the player3D based on the location,
// destination, and time parameters
private void movePlayer(double time) {

    double distanceCovered = (BASE_SPEED + speed) * time;
    double distanceToDestination = location.getDistanceToPoint(destination);
    if (distanceCovered > distanceToDestination)
        distanceCovered = distanceToDestination;

    double angle = location.getAngleToPoint(destination);
    location.moveTo(angle, distanceCovered);

    double xLocation3D = (location.getXLocation() /
                          Globals.GRAPHICS_3D_CONVERSION);
    double yLocation3D = (location.getYLocation() /
                          Globals.GRAPHICS_3D_CONVERSION);
    double zLocation3D = (location.getZLocation() /
                          Globals.GRAPHICS_3D_CONVERSION);

    player3D.setLocation(new DoublePoint(xLocation3D,
                                         yLocation3D,
                                         zLocation3D),
                         (location.getAngleToPoint(destination) - Math.PI/2));
}

// attempts to throw the disc to the selected teammate
private void throwDisc(DoublePoint destination) {

    Disc disc = team.getGame().getDisc();
    double distanceToDestination = location.getDistanceToPoint(destination);
    double angle = location.getAngleToPoint(destination);
    double hVelocity = (disc.DROP_RATE * distanceToDestination) /
        disc.THROWING_HEIGHT;
    disc.fly(hVelocity, angle);
    possessDisc = false;
    team.getGame().stopStallCount();
}

public double getSpeed() {

    return speed;
}

public double getJump() {

    return jump;
}

public double getStrength() {
```

```
    return strength;
}

public double getDexterity() {

    return dexterity;
}

public double getStamina() {

    return stamina;
}

public int getId() {

    return id;
}

public String getName() {

    return name;
}

public boolean hasDisc() {

    return possessDisc;
}

public void setPossessDisc(boolean possessDisc) {

    Disc disc = team.getGame().getDisc();
    this.possessDisc = possessDisc;
    if (possessDisc) {

        disc.setVelocity(0D);
        disc.setToLocation(location);
        disc.setZLocation(Disc.THROWING_HEIGHT);
    }
}

public Player getMarkedPlayer() {

    return markedPlayer;
}

public void setMarkedPlayer(Player markedPlayer) {

    this.markedPlayer = markedPlayer;
}

public String getTeamName() {

    return team.getName();
}

public Team getTeam() {

    return team;
}

public int getCommand() {

    return command;
}

public void setCommand (int command) {

    this.command = command;
}

public DoublePoint getLocation() {
```

```
        return location;
    }

    public DoublePoint getDestination() {

        return destination;
    }

    // updates the player3D location
    public void update() {

        double xLocation3D = (location.getXLocation() /
                              Globals.GRAPHICS_3D_CONVERSION);
        double yLocation3D = (location.getYLocation() /
                              Globals.GRAPHICS_3D_CONVERSION);
        double zLocation3D = (location.getZLocation() /
                              Globals.GRAPHICS_3D_CONVERSION);

        player3D.setLocation(new DoublePoint(xLocation3D,
                                             yLocation3D,
                                             zLocation3D),
                             (location.getAngleToPoint(destination) - Math.PI/2));
    }

    public TransformGroup getTransformGroup() {

        return player3D.getTransformGroup();
    }

    // used by the Replay Controller to save the player's
    // current information
    public Player copyPlayer(Team newTeam) {

        Player clonePlayer = new Player(speed, jump, strength, dexterity, stamina,
                                        id, name, newTeam, new DoublePoint(location));
        clonePlayer.command = command;
        clonePlayer.destinationTimer = destinationTimer;
        clonePlayer.stallCount = stallCount;
        clonePlayer.stallCountRun = stallCountRun;
        clonePlayer.possessDisc = possessDisc;
        clonePlayer.player3D = player3D;
        if (destination == null)
            clonePlayer.destination = null;
        else
            clonePlayer.destination = new DoublePoint(destination);

        return clonePlayer;
    }
}
```

Player3D.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.geometry.*;
import javax.vecmath.*;

//////////////////////////////////////////////////////////////////////
//
//  Player3D.java
//
//  Author: Dean Bolton - 2002
//
//    Class stores all the pertinent information relating to
//    the three-dimensional representation of the player
//
//////////////////////////////////////////////////////////////////////


public class Player3D {

    private DoublePoint location;
    private double angle;
    private int teamId;

    private TransformGroup group = null;
    private TransformGroup outerGroup = null;
    private Transform3D rotationTransform = null;
    private Transform3D translateTransform = null;

    public Player3D(DoublePoint location, double angle, int id) {

        this.location = location;
        this.angle = angle;
        this.teamId = id;

        rotationTransform = new Transform3D();
        translateTransform = new Transform3D();
        outerGroup = new TransformGroup();
        group = new TransformGroup();

        outerGroup.addChild(group);

        outerGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
         outerGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

         group.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
         group.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

        setLocation(location, angle);
    }

    public DoublePoint getLocation() {

        return location;
    }

    public double getAngle() {

        return angle;
    }

    public void setLocation(DoublePoint location, double angle) {

        this.location = location;
        this.angle = angle;
```

```
    translateTransform.set(new Vector3d(location.getXLocation(),
                                        location.getYLocation(),
                                        location.getZLocation()));
    rotationTransform.rotZ(angle);
    updateTransform();
}

private void updateTransform() {

    outerGroup.setTransform(translateTransform);
    group.setTransform(rotationTransform);
}

// sets the appearance of the players
public TransformGroup getTransformGroup() {

    Color3f eColor = new Color3f(Color.white);
    Color3f sColor = new Color3f(Color.black);
    Color3f coneColor = new Color3f(Color.pink);
    Material material = new Material(coneColor, eColor,
                                     coneColor, sColor,
                                     100.0f);

    Appearance appearance = new Appearance();
    material.setLightingEnable(true);
    appearance.setMaterial(material);

    Appearance app = new Appearance();
    ColoringAttributes coloringAttributes;
    if (teamId == Team.RED_TEAM)
        coloringAttributes =
            new ColoringAttributes(new Color3f(Color.red),
                                   ColoringAttributes.FASTEST);
    else
        coloringAttributes =
            new ColoringAttributes(new Color3f(Color.blue),
                                   ColoringAttributes.FASTEST);
    app.setColoringAttributes(coloringAttributes);

    Cylinder bodyShape = new Cylinder(0.05f, 0.15f);
    bodyShape.setAppearance(app);
    Cylinder leftArm = new Cylinder(0.02f, 0.1f);
    leftArm.setAppearance(appearance);
    Cylinder rightArm = new Cylinder(0.02f, 0.1f);
    rightArm.setAppearance(appearance);
    Cylinder leftLeg = new Cylinder(0.02f, 0.1f);
    leftLeg.setAppearance(appearance);
    Cylinder rightLeg = new Cylinder(0.02f, 0.1f);
    rightLeg.setAppearance(appearance);
    Sphere headShape = new Sphere(0.07f, appearance);

    Transform3D translateBody = new Transform3D();
    translateBody.set(new Vector3d(0.0f, 0.0f, 0.175f));
    TransformGroup objTranslateBody = new TransformGroup(translateBody);

     Transform3D translateLeftArm = new Transform3D();
     translateLeftArm.set(new Vector3d(-0.05f, 0.05f, 0.15f));
     TransformGroup objTranslateLeftArm =
         new TransformGroup(translateLeftArm);
     Transform3D translateRightArm = new Transform3D();
     translateRightArm.set(new Vector3d(0.05f, 0.05f, 0.15f));
     TransformGroup objTranslateRightArm =
         new TransformGroup(translateRightArm);
     Transform3D translateLeftLeg = new Transform3D();
     translateLeftLeg.set(new Vector3d(-0.03f, 0.0f, 0.05f));
     TransformGroup objTranslateLeftLeg =
         new TransformGroup(translateLeftLeg);
     Transform3D translateRightLeg = new Transform3D();
     translateRightLeg.set(new Vector3d(0.03f, 0.0f, 0.05f));
     TransformGroup objTranslateRightLeg =
         new TransformGroup(translateRightLeg);
```

```
        Transform3D translateHead = new Transform3D();
        translateHead.set(new Vector3d(0.0f, 0.0f, 0.25f));
        TransformGroup objTranslateHead =
            new TransformGroup(translateHead);

        Transform3D rotate = new Transform3D();
        rotate.rotX(Math.PI/2.0d);
        TransformGroup objRotate = new TransformGroup(rotate);
        Transform3D rotateLeftLeg = new Transform3D();
        rotateLeftLeg.rotX(Math.PI/2.0d);
        TransformGroup objRotateLeftLeg = new TransformGroup(rotateLeftLeg);
        Transform3D rotateRightLeg = new Transform3D();
        rotateRightLeg.rotX(Math.PI/2.0d);
        TransformGroup objRotateRightLeg = new TransformGroup(rotateRightLeg);

        group.addChild(objTranslateBody);
        group.addChild(objTranslateHead);
        group.addChild(objTranslateLeftArm);
        group.addChild(objTranslateRightArm);
        group.addChild(objTranslateLeftLeg);
        group.addChild(objTranslateRightLeg);

        objRotate.addChild(bodyShape);
        objTranslateBody.addChild(objRotate);
        objRotateLeftLeg.addChild(leftLeg);
        objTranslateLeftLeg.addChild(objRotateLeftLeg);
        objRotateRightLeg.addChild(rightLeg);
        objTranslateRightLeg.addChild(objRotateRightLeg);
        objTranslateHead.addChild(headShape);
        objTranslateLeftArm.addChild(leftArm);
        objTranslateRightArm.addChild(rightArm);

        if(teamId == Team.RED_TEAM){

            Transform3D newRot = new Transform3D();
            newRot.rotZ(Math.PI);
            rotationTransform = newRot;
            updateTransform();
        }

       return outerGroup;
    }
}
```

PlayerControlFrame.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.border.*;

////////////////////////////////////////////////////////////////////
//
//  PlayerControlFrame.java
//
//  Author: Dean Bolton - 2002
//
//    Creates a popup frame to display player information
//
////////////////////////////////////////////////////////////////////


public class PlayerControlFrame extends JFrame {

    private JLabel selectedPlayerName;
    private JLabel selectedPlayerSpeed;
    private JLabel selectedPlayerJump;
    private JLabel selectedPlayerStrength;
    private JLabel selectedPlayerDexterity;
    private JLabel selectedPlayerStamina;
    private JLabel selectedPlayerPossessDisc;
    private JLabel selectedPlayerOpponent;
    private JLabel selectedPlayerLocation;
    private JLabel selectedPlayerDestination;

    public PlayerControlFrame(Player player, boolean isVisible) {
        super("Player Controls");

        JPanel contentPane = new JPanel(new GridLayout(0, 1, 5, 5));
        Border controlBorder = new CompoundBorder(new EtchedBorder(),
                                                  new EmptyBorder(5, 5, 5, 5));
        contentPane.setBorder(controlBorder);
        this.setContentPane(contentPane);

        Font playerHeaderFont = new Font("Dialog", Font.BOLD, 12);
        JLabel selectedPlayerLabel = new JLabel("Player Information", JLabel.LEFT);
        selectedPlayerLabel.setFont(playerHeaderFont);
        this.getContentPane().add(selectedPlayerLabel);

        Font playerFont = new Font("Dialog", Font.BOLD, 10);
        JLabel selectedPlayerName = new JLabel("click a player to view",
                                               JLabel.LEFT);
        selectedPlayerName.setFont(playerFont);
        this.getContentPane().add(selectedPlayerName);

        selectedPlayerSpeed = new JLabel("", JLabel.LEFT);
        selectedPlayerSpeed.setFont(playerFont);
        this.getContentPane().add(selectedPlayerSpeed);

        selectedPlayerJump = new JLabel("", JLabel.LEFT);
        selectedPlayerJump.setFont(playerFont);
        this.getContentPane().add(selectedPlayerJump);

        selectedPlayerStrength = new JLabel("", JLabel.LEFT);
        selectedPlayerStrength.setFont(playerFont);
        this.getContentPane().add(selectedPlayerStrength);

        selectedPlayerDexterity = new JLabel("", JLabel.LEFT);
        selectedPlayerDexterity.setFont(playerFont);
        this.getContentPane().add(selectedPlayerDexterity);

        selectedPlayerStamina = new JLabel("", JLabel.LEFT);
```

```
        selectedPlayerStamina.setFont(playerFont);
        this.getContentPane().add(selectedPlayerStamina);

        selectedPlayerPossessDisc = new JLabel("", JLabel.LEFT);
        selectedPlayerPossessDisc.setFont(playerFont);
        this.getContentPane().add(selectedPlayerPossessDisc);

        selectedPlayerOpponent = new JLabel("", JLabel.LEFT);
        selectedPlayerOpponent.setFont(playerFont);
        this.getContentPane().add(selectedPlayerOpponent);

        selectedPlayerLocation = new JLabel("", JLabel.LEFT);
        selectedPlayerLocation.setFont(playerFont);
        this.getContentPane().add(selectedPlayerLocation);

        selectedPlayerDestination = new JLabel("", JLabel.LEFT);
        selectedPlayerDestination.setFont(playerFont);
        this.getContentPane().add(selectedPlayerDestination);

        JLabel empty = new JLabel("", JLabel.LEFT);
        empty.setFont(playerFont);
        this.getContentPane().add(empty);
        empty = new JLabel("", JLabel.LEFT);
        empty.setFont(playerFont);
        this.getContentPane().add(empty);
        empty = new JLabel("", JLabel.LEFT);
        empty.setFont(playerFont);
        this.getContentPane().add(empty);
    }

    public void displaySelectedPlayer(Player player) {

        if (player != null) {

            if (player.getId() == Team.RED_TEAM)
                selectedPlayerName.setForeground(Color.red);
            else
                selectedPlayerName.setForeground(Color.blue);

            selectedPlayerName.setText("Name: " + player.getName());
            selectedPlayerSpeed.setText("Speed: " + player.getSpeed());
            selectedPlayerJump.setText("Jump: " + player.getJump());
            selectedPlayerStrength.setText("Strength: " + player.getStrength());
            selectedPlayerDexterity.setText("Dexterity: " + player.getDexterity());
            selectedPlayerStamina.setText("Stamina: " + player.getStamina());
            selectedPlayerPossessDisc.setText("Has Disc: " +
                                              player.hasDisc());
            selectedPlayerOpponent.setText("Opponent: " +
                                           player.getMarkedPlayer().getId());

            DoublePoint location = player.getLocation();
            DoublePoint destination = player.getDestination();

            selectedPlayerLocation.setText("Location X: " +
                                           (int) location.getXLocation() +
                                           " Y: " +
                                           (int) location.getYLocation());
            selectedPlayerDestination.setText("Destination X: " +
                                              (int) destination.getXLocation() +
                                              " Y: " +
                                              (int) destination.getYLocation());
        } else {

            selectedPlayerName.setForeground(new Color(100, 100, 150));
            selectedPlayerName.setText("click a player to view");

            selectedPlayerSpeed.setText("");
            selectedPlayerJump.setText("");
            selectedPlayerStrength.setText("");
            selectedPlayerDexterity.setText("");
            selectedPlayerStamina.setText("");
```

```
            selectedPlayerPossessDisc.setText("");
            selectedPlayerOpponent.setText("");
            selectedPlayerLocation.setText("");
            selectedPlayerDestination.setText("");
        }
    }
}
```

PlayingField.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

////////////////////////////////////////////////////////////////////////
//
//  PlayingField.java
//
//  Author: Dean Bolton - 2002
//
//    Controls the two-dimensional representation of the game
//    in the FieldControlFrame
//
////////////////////////////////////////////////////////////////////////


public class PlayingField extends JPanel implements Runnable, MouseListener {

    public static final double FIELD_SIZE_MULTIPLIER = 5D;
    public static final int SLEEP_TIME = 50;

    private Ultimate uf;
    private Image fieldBackgroundImage;
    private Thread refreshThread;
    private DoublePoint centerPoint;
    private boolean isApplet;

    public PlayingField(Ultimate uf, boolean isApplet) {
        super();

        this.uf = uf;
        this.isApplet = isApplet;

        addMouseListener(this);
        setBackground(Color.green);
        start();
    }

    private void start() {

        if ((refreshThread == null) || (!refreshThread.isAlive())) {

            refreshThread = new Thread(this, "Playing Field");
            refreshThread.start();
        }
    }

    public void run() {

        while (true) {

            try {

                refreshThread.sleep(SLEEP_TIME);
            } catch (InterruptedException e) {}

            repaint();
        }
    }

    // method to refresh the two-dimensional graphics
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        Dimension panelSize = getSize();
        if (centerPoint == null)
```

```
        centerPoint = new DoublePoint(panelSize.getWidth() / 2D,
                                      panelSize.getHeight() / 2D);

    if (fieldBackgroundImage == null)
        fieldBackgroundImage = createFieldBackground(panelSize);
    g.drawImage(fieldBackgroundImage, 0, 0, this);

    g.setColor(Color.red);
    drawTeam(uf.getGame().getRedTeam(), g);

    g.setColor(Color.blue);
    drawTeam(uf.getGame().getBlueTeam(), g);

    drawDisc(g);
}

// draws the disc in the playing field
private void drawDisc(Graphics g) {

    g.setColor(Color.yellow);
    Disc disc = uf.getGame().getDisc();
    int discX = (int) ((disc.getXLocation() * FIELD_SIZE_MULTIPLIER) +
                       centerPoint.getXLocation());
    int discY = (int) ((disc.getYLocation() * FIELD_SIZE_MULTIPLIER) +
                       centerPoint.getYLocation());
    g.fillOval(discX - 3, discY - 3, 6, 6);
}

// draws the red and blue teams on the playing field
private void drawTeam(Team team, Graphics g) {

    for (int x=0; x<7; x++) {

        Player player = team.getPlayer(x);
        int playerX =
            (int) ((player.getLocation().getXLocation() *
                    FIELD_SIZE_MULTIPLIER) +
                   centerPoint.getXLocation());
        int playerY =
            (int) ((player.getLocation().getYLocation() *
                    FIELD_SIZE_MULTIPLIER) +
                   centerPoint.getYLocation());
        g.fillOval(playerX - 5, playerY - 5, 10, 10);
    }
}

// draws the background of the playing field
private Image createFieldBackground(Dimension panelSize) {

    Image buff = createImage((int) panelSize.getWidth(),
                             (int) panelSize.getHeight());
    Graphics buffG = buff.getGraphics();

    Image topBuff = createImage((int) panelSize.getWidth(),
                                (int) panelSize.getHeight());
    Graphics topBuffG = topBuff.getGraphics();

    int fieldWidth =
        (int) (Globals.BASE_FIELD_WIDTH * FIELD_SIZE_MULTIPLIER);
    int fieldLength =
        (int) (Globals.BASE_FIELD_LENGTH * FIELD_SIZE_MULTIPLIER);
    int fieldBoundLeftX = (int) centerPoint.getXLocation() -
        (fieldWidth / 2);
    int fieldBoundRightX = (int) centerPoint.getXLocation() +
        (fieldWidth / 2);
    int fieldBoundTopY = (int) centerPoint.getYLocation() -
        (fieldLength / 2);
    int fieldBoundBottomY = (int) centerPoint.getYLocation() +
        (fieldLength / 2);

    topBuffG.setColor(Color.green);
```

```
    topBuffG.fillRect(0, 0,
                      (int) panelSize.getWidth(),
                      (int) panelSize.getHeight());

    topBuffG.setColor(Color.white);
    topBuffG.drawRect(fieldBoundLeftX, fieldBoundTopY,
                      fieldWidth, fieldLength);

    int endZoneLength =
        (int) (Globals.BASE_ENDZONE_LENGTH * FIELD_SIZE_MULTIPLIER);

    topBuffG.drawRect(fieldBoundLeftX, fieldBoundTopY,
                      fieldWidth, endZoneLength);
    Graphics topEndZoneClip = topBuffG.create(fieldBoundLeftX,
                                              fieldBoundTopY,
                                              fieldWidth,
                                              endZoneLength);
    for (int x = 0 - (endZoneLength / 2); x < fieldWidth;
         x += (endZoneLength / 2)) {

        topEndZoneClip.drawLine(x, 0,
                                x + (endZoneLength / 2),
                                endZoneLength);
    }
    topEndZoneClip.dispose();

    topBuffG.drawRect(fieldBoundLeftX, fieldBoundBottomY-endZoneLength,
                      fieldWidth, endZoneLength);
    Graphics bottomEndZoneClip =
        topBuffG.create(fieldBoundLeftX,
                        fieldBoundBottomY-endZoneLength,
                        fieldWidth,
                        endZoneLength);
    for (int x = 0 - (endZoneLength / 2); x < fieldWidth;
         x += (endZoneLength / 2)) {

        bottomEndZoneClip.drawLine(x, 0,
                                   x + (endZoneLength / 2),
                                   endZoneLength);
    }
    bottomEndZoneClip.dispose();

    topBuffG.setColor(Color.orange);
    topBuffG.fillOval(fieldBoundLeftX - 2, fieldBoundTopY - 2, 5, 5);
    topBuffG.fillOval(fieldBoundRightX - 2, fieldBoundTopY - 2, 5, 5);
    topBuffG.fillOval(fieldBoundLeftX - 2,
                      fieldBoundTopY + endZoneLength - 2,
                      5, 5);
    topBuffG.fillOval(fieldBoundRightX - 2,
                      fieldBoundTopY + endZoneLength - 2 + 1,
                      5, 5);
    topBuffG.fillOval(fieldBoundLeftX - 2, fieldBoundBottomY - 2, 5, 5);
    topBuffG.fillOval(fieldBoundRightX - 2, fieldBoundBottomY - 2, 5, 5);
    topBuffG.fillOval(fieldBoundLeftX - 2,
                      fieldBoundBottomY - endZoneLength - 2,
                      5, 5);
    topBuffG.fillOval(fieldBoundRightX - 2,
                      fieldBoundBottomY - endZoneLength - 2,
                      5, 5);

    buffG.drawImage(topBuff, 0, 0, this);

    return buff;
}

// used to find the back of either endzone
public static double getEndZoneBoundary(boolean side) {

    if (side)
        return (Globals.BASE_PLAYING_LENGTH / 2D);
    else
```

```
        return 0D - (Globals.BASE_PLAYING_LENGTH / 2D);
}

// check to see if the location is within a given endzone
public static boolean isEndZone(boolean side, DoublePoint location) {

    boolean result = false;
    if (side) {

        if (location.getYLocation() <=
            0D - (Globals.BASE_PLAYING_LENGTH / 2D)) {

            if (location.getYLocation() >=
                0D - (Globals.BASE_FIELD_LENGTH / 2D)) {

                if (isInBounds(location))
                    result = true;
            }
        }
    } else {

        if (location.getYLocation() >=
            (Globals.BASE_PLAYING_LENGTH / 2D)) {

            if (location.getYLocation() <=
                (Globals.BASE_FIELD_LENGTH / 2D)) {

                if (isInBounds(location))
                    result = true;
            }
        }
    }
    return result;
}

// checks to see whether a location is in bounds
public static boolean isInBounds(DoublePoint location) {

    if (location.getYLocation() >=
        0D - (Globals.BASE_FIELD_LENGTH / 2D)) {

        if (location.getYLocation() <=
            (Globals.BASE_FIELD_LENGTH / 2D)) {

            if (location.getXLocation() >=
                0D - (Globals.BASE_FIELD_WIDTH / 2D)) {

                if (location.getXLocation() <=
                    (Globals.BASE_FIELD_WIDTH / 2D)) {

                    return true;
                }
            }
        }
    }
    return false;
}

public void mouseClicked(MouseEvent e) {

    double x = e.getX();
    double y = e.getY();

    Dimension panelSize = getSize();
    if (centerPoint == null)
        centerPoint = new DoublePoint(panelSize.getWidth() / 2D,
                                      panelSize.getHeight() / 2D);
    x = (x - centerPoint.getXLocation()) / FIELD_SIZE_MULTIPLIER;
    y = (y - centerPoint.getYLocation()) / FIELD_SIZE_MULTIPLIER;
    DoublePoint clicked = new DoublePoint(x, y);
```

```
        Team redTeam = uf.getGame().getRedTeam();
        Team blueTeam = uf.getGame().getBlueTeam();
        Player closestToClick = null;
        double clickDistance = Double.MAX_VALUE;

        for (int i=0; i<7; i++) {

            double newDistance =
                redTeam.getPlayer(i).getLocation().getDistanceToPoint(clicked);
            if (newDistance < clickDistance) {

                closestToClick = redTeam.getPlayer(i);
                clickDistance = newDistance;
            }

            newDistance =
                blueTeam.getPlayer(i).getLocation().getDistanceToPoint(clicked);
            if (newDistance < clickDistance) {

                closestToClick = blueTeam.getPlayer(i);
                clickDistance = newDistance;
            }
        }

        if (clickDistance < 2D)
            uf.getPlayerControl().displaySelectedPlayer(closestToClick);
    }

    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}
```

ReplayController.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.util.*;

//////////////////////////////////////////////////////////////////////
//
//  ReplayController.java
//
//  Author: Dean Bolton - 2002
//
//     Class that stores all of the replay information.  Creates
//     an ArrayList to store the game information.  Updates the
//     game based on clicks in the SimulationControlFrame
//
//////////////////////////////////////////////////////////////////////


public class ReplayController {

    private Ultimate uf;
    private SimulationControlFrame simulationControl;
    private Game activeGame;
    private ArrayList recordedGames;

    public ReplayController(Ultimate uf, Game activeGame) {

        this.uf = uf;
        this.activeGame = activeGame;
        recordedGames = new ArrayList();
    }

    public void setSimulationControlFrame(SimulationControlFrame simulationControl) {

        this.simulationControl = simulationControl;
    }

    public void recordGame() {

        recordedGames.add(activeGame.copyGame());
        if (recordedGames.size() > Globals.MAX_RECORDED_GAMES)
            recordedGames.remove(0);
    }

    // returns the state of the game to the beginning or thirty (30)
    // seconds ago, whichever is more recent
    public void restart() {

        if (recordedGames.size() > 0) {

            activeGame = (Game) recordedGames.get(0);
            uf.setGame(activeGame);
        }
        activeGame.pauseGame();
        simulationControl.markActiveButton(SimulationControlFrame.RESTART);
    }

    // moves the game back one second in time
    public void rewind() {

        if (recordedGames.contains(activeGame)) {

            int index = recordedGames.indexOf(activeGame);
            if (index > 0) {

                activeGame = (Game) recordedGames.get(index - 1);
```

```
            uf.setGame(activeGame);
        }
    } else {

        if (recordedGames.size() > 0) {

            activeGame = (Game) recordedGames.get(recordedGames.size() - 1);
            uf.setGame(activeGame);
        }
    }
    activeGame.pauseGame();
    simulationControl.markActiveButton(SimulationControlFrame.REWIND);
}

// resumes the game based on the current conditions
public void play() {

    activeGame.runGame();
    if (recordedGames.contains(activeGame)) {

        int index = recordedGames.indexOf(activeGame);
        int size = recordedGames.size();

        for (int i = size; i > index+1; i--) {

            recordedGames.remove(i-1);
        }
    }
    simulationControl.markActiveButton(SimulationControlFrame.PLAY);
}

// pauses the game at the current state
public void pause() {

    activeGame.pauseGame();
    simulationControl.markActiveButton(SimulationControlFrame.PAUSE);
}

// if the game has been rewound, increments the game by one
// second in time
public void fastForward() {

    if (recordedGames.contains(activeGame)) {

        int index = recordedGames.indexOf(activeGame);
        if (index < (recordedGames.size() - 1)) {

            activeGame = (Game) recordedGames.get(index + 1);
            uf.setGame(activeGame);
        }
    }
    activeGame.pauseGame();
    simulationControl.markActiveButton(SimulationControlFrame.FAST_FORWARD);
}

// if the game has been rewound, brings the game back
// to the most recent state
public void skipAhead() {

    if (recordedGames.contains(activeGame)) {

        activeGame = (Game) recordedGames.get(recordedGames.size() - 1);
        uf.setGame(activeGame);
    }
    activeGame.pauseGame();
    simulationControl.markActiveButton(SimulationControlFrame.SKIP_AHEAD);
}
}
```

SimulationControlFrame.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.net.*;

//////////////////////////////////////////////////////////////////////
//
//  SimulationControlFrame.java
//
//  Author: Dean Bolton - 2002
//
//    Popup frame that control the replay features of the game
//
//////////////////////////////////////////////////////////////////////


public class SimulationControlFrame extends JFrame implements MouseListener {

    public static final int ERROR = 0;
    public static final int RESTART = 1;
    public static final int REWIND = 2;
    public static final int PLAY = 3;
    public static final int PAUSE = 4;
    public static final int FAST_FORWARD = 5;
    public static final int SKIP_AHEAD = 6;

    private ReplayController replayController;
    private JLabel restartLabel;
    private JLabel rewindLabel;
    private JLabel playLabel;
    private JLabel pauseLabel;
    private JLabel fastForwardLabel;
    private JLabel skipAheadLabel;

    public SimulationControlFrame(ReplayController replayController,
                                  URL codeBase, boolean isApplet) {
        super("Simulator Controls");

        this.replayController = replayController;

        setDefaultCloseOperation(WindowConstants.HIDE_ON_CLOSE);
        if (isApplet)
            setSize(250, 120);
        else
            setSize(250, 100);

        setLocation(50, 560);

        JPanel contentPane = new JPanel(new FlowLayout());
        contentPane.setBackground(Color.white);
        contentPane.setDoubleBuffered(true);
        this.setContentPane(contentPane);

        ImageIcon restartIcon = null;
        ImageIcon rewindIcon = null;
        ImageIcon playIcon = null;
        ImageIcon pauseIcon = null;
        ImageIcon fastForwardIcon = null;
        ImageIcon skipAheadIcon = null;

        // creates the replay buttons
        if (codeBase != null) {

            try {

                restartIcon = new ImageIcon(new URL(codeBase,
```

```
                                                "images/restart.gif"));
            rewindIcon = new ImageIcon(new URL(codeBase,
                                               "images/rewind.gif"));
            playIcon = new ImageIcon(new URL(codeBase,
                                                "images/play.gif"));
            pauseIcon = new ImageIcon(new URL(codeBase,
                                              "images/pause.gif"));
            fastForwardIcon = new ImageIcon(new URL(codeBase,
                                                    "images/fast_forward.gif"));
            skipAheadIcon = new ImageIcon(new URL(codeBase,
                                                  "images/skip_ahead.gif"));
        } catch (MalformedURLException e) {

            System.out.println(e.toString());
        }
    } else {

        restartIcon = new ImageIcon("images/restart.gif");
        rewindIcon = new ImageIcon("images/rewind.gif");
        playIcon = new ImageIcon("images/play.gif");
        pauseIcon = new ImageIcon("images/pause.gif");
        fastForwardIcon = new ImageIcon("images/fast_forward.gif");
        skipAheadIcon = new ImageIcon("images/skip_ahead.gif");
    }

    restartLabel = new JLabel(restartIcon);
    restartLabel.setBackground(Color.red);
    restartLabel.setOpaque(true);
    restartLabel.addMouseListener(this);
    this.getContentPane().add(restartLabel);

    rewindLabel = new JLabel(rewindIcon);
    rewindLabel.setBackground(Color.red);
    rewindLabel.setOpaque(true);
    rewindLabel.addMouseListener(this);
    this.getContentPane().add(rewindLabel);

    playLabel = new JLabel(playIcon);
    playLabel.setBackground(Color.red);
    playLabel.setOpaque(true);
    playLabel.addMouseListener(this);
    this.getContentPane().add(playLabel);

    pauseLabel = new JLabel(pauseIcon);
    pauseLabel.setBackground(Color.red);
    pauseLabel.setOpaque(true);
    pauseLabel.addMouseListener(this);
    this.getContentPane().add(pauseLabel);

    fastForwardLabel = new JLabel(fastForwardIcon);
    fastForwardLabel.setBackground(Color.red);
    fastForwardLabel.setOpaque(true);
    fastForwardLabel.addMouseListener(this);
    this.getContentPane().add(fastForwardLabel);

    skipAheadLabel = new JLabel(skipAheadIcon);
    skipAheadLabel.setBackground(Color.red);
    skipAheadLabel.setOpaque(true);
    skipAheadLabel.addMouseListener(this);
    this.getContentPane().add(skipAheadLabel);
}

// appropriately colors the active button
public void markActiveButton(int button) {

    if (button == RESTART)
        restartLabel.setBackground(Color.green);
    else
        restartLabel.setBackground(Color.red);

    if (button == REWIND)
```

```
            rewindLabel.setBackground(Color.green);
        else
            rewindLabel.setBackground(Color.red);

        if (button == PLAY)
            playLabel.setBackground(Color.green);
        else
            playLabel.setBackground(Color.red);

        if (button == PAUSE)
            pauseLabel.setBackground(Color.green);
        else
            pauseLabel.setBackground(Color.red);

        if (button == FAST_FORWARD)
            fastForwardLabel.setBackground(Color.green);
        else
            fastForwardLabel.setBackground(Color.red);

        if (button == SKIP_AHEAD)
            skipAheadLabel.setBackground(Color.green);
        else
            skipAheadLabel.setBackground(Color.red);
    }

    public ReplayController getReplayController() {

        return replayController;
    }

    // implements the mouse listener to control the Replay Controller
    public void mousePressed(MouseEvent e) {

        Object object = e.getSource();
        if (object == restartLabel)
            replayController.restart();
        else if (object == rewindLabel)
            replayController.rewind();
        else if (object == playLabel)
            replayController.play();
        else if (object == pauseLabel)
            replayController.pause();
        else if (object == fastForwardLabel)
            replayController.fastForward();
        else if (object == skipAheadLabel)
            replayController.skipAhead();
    }

    public void mouseClicked(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}
```

SimulationMenuBar.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

//////////////////////////////////////////////////////////////////////
//
//  SimulationMenuBar.java
//
//  Author: Dean Bolton - 2002
//
//    Creates the menu bar for the main frame of the simulation
//
//////////////////////////////////////////////////////////////////////


public class SimulationMenuBar extends JMenuBar implements ActionListener {

    private JMenuItem restartMenuItem, rewindMenuItem, playMenuItem;
    private JMenuItem pauseMenuItem, fastForwardMenuItem, skipAheadMenuItem;
    private JMenuItem loadMenuItem, saveMenuItem, exitMenuItem;
    private JMenuItem simulationMenuItem, cameraMenuItem, fieldMenuItem;
    private JMenuItem playerMenuItem, teamMenuItem, aboutMenuItem;

    private SimulationControlFrame simulationControl;
    private CameraControlFrame cameraControl;
    private FieldControlFrame fieldControl;
    private PlayerControlFrame playerControl;
    //    private TeamControlFrame teamControl;
    private AboutFrame about;

    public SimulationMenuBar(SimulationControlFrame simulationControl,
                             CameraControlFrame cameraControl,
                             FieldControlFrame fieldControl,
                             PlayerControlFrame playerControl,
                             //      TeamControlFrame teamControl,
                             AboutFrame about) {

        // Use JMenuBar constructor.
        super();
        JPopupMenu.setDefaultLightWeightPopupEnabled(false);

        // Initialize data members.
        this.simulationControl = simulationControl;
        this.cameraControl = cameraControl;
        this.fieldControl = fieldControl;
        this.playerControl = playerControl;
        //      this.teamControl = teamControl;
        this.about = about;

        //  Simulation Menu with "S" quick key
        JMenu simMenu = new JMenu("Simulation");
        simMenu.setMnemonic(KeyEvent.VK_S);
        this.add(simMenu);

        //  Panel Menu with "P" quick key
        JMenu panelMenu = new JMenu("Panels");
        panelMenu.setMnemonic(KeyEvent.VK_P);
        this.add(panelMenu);

        //  Help menu with "H" quick key
        JMenu helpMenu = new JMenu("Help");
        helpMenu.setMnemonic(KeyEvent.VK_H);
                this.add(Box.createHorizontalGlue());
        this.add(helpMenu);

        //  Adds "Restart", "Rewind", "Play", "Pause", "Fast Forward",
```

```
        //  "Skip Ahead", "Load", "Save", and "Exit" menu items
        restartMenuItem = new JMenuItem("Restart");
        restartMenuItem.addActionListener(this);
        simMenu.add(restartMenuItem);
        rewindMenuItem = new JMenuItem("Rewind");
        rewindMenuItem.addActionListener(this);
        simMenu.add(rewindMenuItem);
        playMenuItem = new JMenuItem("Play");
        playMenuItem.addActionListener(this);
        simMenu.add(playMenuItem);
        pauseMenuItem = new JMenuItem("Pause");
        pauseMenuItem.addActionListener(this);
        simMenu.add(pauseMenuItem);
        fastForwardMenuItem = new JMenuItem("Fast Forward");
        fastForwardMenuItem.addActionListener(this);
        simMenu.add(fastForwardMenuItem);
        skipAheadMenuItem = new JMenuItem("Skip Ahead");
        skipAheadMenuItem.addActionListener(this);
        simMenu.add(skipAheadMenuItem);
        simMenu.addSeparator();

//  These features are not available in current version
//      loadMenuItem = new JMenuItem("Load");
//       loadMenuItem.addActionListener(this);
//      simMenu.add(loadMenuItem);
//      saveMenuItem = new JMenuItem("Save");
//       saveMenuItem.addActionListener(this);
//      simMenu.add(saveMenuItem);
//      simMenu.addSeparator();

        exitMenuItem = new JMenuItem("Exit");
        exitMenuItem.addActionListener(this);
        simMenu.add(exitMenuItem);

        //  Adds "Panels" menu items
        simulationMenuItem = new JMenuItem("Simulator Controls");
        simulationMenuItem.addActionListener(this);
        panelMenu.add(simulationMenuItem);
        cameraMenuItem = new JMenuItem("Camera Controls");
        cameraMenuItem.addActionListener(this);
        panelMenu.add(cameraMenuItem);
        fieldMenuItem = new JMenuItem("Field Controls");
        fieldMenuItem.addActionListener(this);
        panelMenu.add(fieldMenuItem);
        playerMenuItem = new JMenuItem("Player Controls");
        playerMenuItem.addActionListener(this);
        panelMenu.add(playerMenuItem);
        //      teamMenuItem = new JMenuItem("Team Controls");
        //      teamMenuItem.addActionListener(this);
        //      panelMenu.add(teamMenuItem);

        //  Adds "About" menu item
        aboutMenuItem = new JMenuItem("About");
        aboutMenuItem.addActionListener(this);
        helpMenu.add(aboutMenuItem);
    }

    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();

        ReplayController rc = simulationControl.getReplayController();

        if (source == restartMenuItem) {
            rc.restart();
        }
        else if (source == rewindMenuItem) {
            rc.rewind();
        }
        else if (source == playMenuItem) {
            rc.play();
        }
```

```
        else if (source == pauseMenuItem) {
            rc.pause();
        }
        else if (source == fastForwardMenuItem) {
            rc.fastForward();
        }
        else if (source == skipAheadMenuItem) {
            rc.skipAhead();
        }
        /*
        else if (source == loadMenuItem) {
        }
        else if (source == saveMenuItem) {
        }
        */
        else if (source == exitMenuItem) {
            System.exit(0);
        }
        else if (source == simulationMenuItem) {
            simulationControl.setVisible(true);
        }
        else if (source == cameraMenuItem) {
            cameraControl.setVisible(true);
        }
        else if (source == fieldMenuItem) {
            fieldControl.setVisible(true);
        }
        else if (source == playerMenuItem) {
            playerControl.setVisible(true);
        }
        /*
        else if (source == teamMenuItem) {
            teamControl.setVisible(true);
        }
        */
        else if (source == aboutMenuItem) {
            about.setVisible(true);
        }
    }
}
```

Team.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

////////////////////////////////////////////////////////////////////////
//
//  Team.java
//
//  Author: Dean Bolton - 2002
//
//    Implements the team functionality and initializes the
//    players
//
////////////////////////////////////////////////////////////////////////


public class Team {

    public static final int RED_TEAM = 1;
    public static final int BLUE_TEAM = 2;

    private String name = "";
    private int id = 0;
    private Player[] players = null;
    private boolean possession = false;
    private int score = 0;
    private DoublePoint[] pullPositions;
    private Game game = null;
    private boolean side = false;

    public Team(int id, String name, boolean possession,
                boolean side, Game game) {

        this.id = id;
        this.name = name;
        this.possession = possession;
        this.game = game;
        this.side = side;

        players = new Player[7];

        DoublePoint startLocation = new DoublePoint(21D, -10D);
        if (side)
            startLocation = new DoublePoint(-21D, -20D);

        // creates new players at starting locations on the sideline
        for (int i=0; i<7; i++) {

            String playerName = "player " + (i + 1);
            players[i] = new Player(0D, 0D, 0D, 0D, 0D, i, playerName, this,
                                    new DoublePoint(startLocation));
            startLocation.setYLocation(startLocation.getYLocation() + 5);
        }

        determinePullPositions();
    }

    public Game getGame() {

        return game;
    }

    public int getId() {

        return id;
    }
```

```
public String getName() {

    return name;
}

public boolean hasPossession() {

    return possession;
}

public void setPossession(boolean possession) {

    this.possession = possession;
}

// updates the players after a turnover
public void changePossession() {

    possession = !possession;
    for (int i=0; i<7; i++) {

        if (possession) {

            if (players[i].hasDisc())
                players[i].setCommand(Player.THROW);
            else
                players[i].setCommand(Player.OFFENSE);
        } else {

            players[i].setCommand(Player.DEFEND);
        }
    }
}

public boolean getPlayingFieldSide() {

    return side;
}

public void setPlayingFieldSide(boolean side) {

    this.side = side;
}

public Player getPossessingPlayer() {

    Player possessor = null;
    for (int i=0; i<7; i++) {

        if (players[i].hasDisc())
            possessor = players[i];
    }
    return possessor;
}

public void clearPlayerPossession() {

    for (int i=0; i<7; i++) {

        players[i].setPossessDisc(false);
    }
}

public Player getPlayer(int index) {

    return players[index];
}

public int getScore() {
```

```
        return score;
    }

    public void incrementScore() {

        score++;
    }

    public void clearScore() {

        score = 0;
    }

    public boolean atPullPositions() {

        boolean result = true;
        for (int i=0; i<7; i++) {

            if (!players[i].getLocation().reachedPoint(pullPositions[i]))
                result = false;
        }
        return result;
    }

    public void goToPullPositions() {

        for (int i=0; i<7; i++) {

            players[i].moveTo(pullPositions[i]);
        }
    }

    public void pull(boolean receiving) {

        Player closestToDisc = players[0];
        if (receiving) {

            double discDistance =
                players[0].getLocation().getDistanceToPoint(game.getDisc().getLocation());
            for (int i=1; i<7; i++) {

                double newDistance =

players[i].getLocation().getDistanceToPoint(game.getDisc().getLocation());
                if (newDistance < discDistance) {

                    closestToDisc = players[i];
                    discDistance = newDistance;
                }
            }
        }

        for (int i=0; i<7; i++) {

            if (!receiving) {

                if (players[i].hasDisc())
                    players[i].setCommand(Player.PULL);
                else
                    players[i].setCommand(Player.DEFEND);
            } else {

                if (players[i] == closestToDisc)
                    players[i].setCommand(Player.CHASE_DISC);
                else
                    players[i].setCommand(Player.OFFENSE);
            }
        }
    }

    // used to have closest player on offense pick up the disc
```

```
    public void fetchDisc() {

        Player closestToDisc = players[0];
        double discDistance =
            players[0].getLocation().getDistanceToPoint(game.getDisc().getLocation());
        for (int i=1; i<7; i++) {

            double newDistance =
                players[i].getLocation().getDistanceToPoint(game.getDisc().getLocation());
            if (newDistance < discDistance) {

                closestToDisc = players[i];
                discDistance = newDistance;
            }
        }
        closestToDisc.setCommand(Player.CHASE_DISC);
    }

    // selects positions at which to pull the disc
    public void determinePullPositions() {

        DoublePoint pull =
            new DoublePoint((int) -(Globals.BASE_FIELD_WIDTH / 2D),
                            (int) (Globals.BASE_PLAYING_LENGTH / 2D));

        if (side)
            pull.setYLocation(-pull.getYLocation());

        pullPositions = new DoublePoint[7];
        for (int i=0; i<7; i++) {

            pullPositions[i] = new DoublePoint(pull);
            pull.setXLocation(pull.getXLocation() + 6);
        }
    }

    public void stopPlayers() {

        for (int i=0; i<7; i++) {

            players[i].setCommand(Player.STOP);
        }
    }

    // simple method to determine which player is open.  calculates
    // a weighted probability based on the separation between the
    // offensive player and the defensive player.  a random player
    // is selected using that weighted probability
    public Player findOpenReceiver(Player passer) {

        Player result = null;
        Player temp = null;
        double totalProbability = 0D;

        for (int i=0; i<7; i++) {

            temp = players[i];
            if (temp != passer) {

                double dist =
temp.getLocation().getDistanceToPoint(temp.getMarkedPlayer().getLocation());
                dist = dist * dist * dist;
                totalProbability += dist;
            }
        }

        double r = Math.random() * totalProbability;
        for (int i=0; i<7; i++) {

            temp = players[i];
            if (temp != passer) {
```

```
                double probability =
temp.getLocation().getDistanceToPoint(temp.getMarkedPlayer().getLocation());
                probability = probability * probability * probability;
                if (result == null) {

                    if (r < probability)
                        result = temp;
                    else
                        r -= probability;
                }
            }
        }
        return result;
    }

    public void setMarkedPlayers() {

        Team oppositeTeam = null;
        if (id == RED_TEAM)
            oppositeTeam = game.getBlueTeam();
        else
            oppositeTeam = game.getRedTeam();

        for (int i=0; i<7; i++) {
            players[i].setMarkedPlayer(oppositeTeam.getPlayer(i));
        }
    }

    // used by the Replay Controller to save the team information
    public Team copyTeam(Game newGame) {

        Team cloneTeam = new Team(id, name, possession, side, newGame);
        cloneTeam.score = score;

        for (int i=0; i<7; i++) {

            cloneTeam.pullPositions[i] = pullPositions[i];
            cloneTeam.players[i] = players[i].copyPlayer(cloneTeam);
        }
        return cloneTeam;
    }
}
```

Ultimate.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.lang.*;
import java.net.*;
import javax.media.j3d.*;

//////////////////////////////////////////////////////////////////////
//
//  Ultimate.java
//
//  Author: Dean Bolton - 2002
//
//    Main class to set up the simulator, which calls a lot of the
//    other classes
//
//////////////////////////////////////////////////////////////////////


public class Ultimate extends JPanel implements Runnable {

    // Game play time pulse in milliseconds
    private static final double TIME_PULSE = 50D;

    private Thread gameThread = null;
    private Game game = null;
    private ReplayController replayController = null;
    private double recordTimer = 0D;
    private FieldControlFrame fieldControl = null;
    private CameraControlFrame cameraControl = null;
    private PlayerControlFrame playerControl = null;
    //    private TeamControlFrame teamControl = null;
    private SimulationControlFrame simulationControl = null;
    private SimulationMenuBar simulationMenuBar = null;
    private World3D world3D = null;

    public Ultimate(URL codeBase, boolean isApplet) {
        super(new BorderLayout());

        game = new Game(true);

        fieldControl = new FieldControlFrame(this, isApplet, codeBase, game);
        playerControl = new PlayerControlFrame(null, false);
        //      teamControl = new TeamControlFrame();

        replayController = new ReplayController(this, game);
        replayController.recordGame();

        simulationControl =
            new SimulationControlFrame(replayController, codeBase, isApplet);
        replayController.setSimulationControlFrame(simulationControl);

        world3D = new World3D(game);
        add("Center", world3D.getCanvas3D());

        cameraControl = new CameraControlFrame(world3D);

        AboutFrame about = new AboutFrame();
        simulationMenuBar = new SimulationMenuBar(simulationControl,
                                                  cameraControl,
                                                  fieldControl,
                                                  playerControl,
                                                  //  teamControl,
                                                  about);
```

```
    setVisible(true);

    start();
}

public JMenuBar getMenu() {

    return simulationMenuBar;
}

public Game getGame() {

    return game;
}

public FieldControlFrame getFieldControl() {

    return fieldControl;
}

/*
public TeamControlFrame getTeamControl() {

    return teamControl;
}
*/

public PlayerControlFrame getPlayerControl() {

    return playerControl;
}

public SimulationControlFrame getSimulationControl() {

    return simulationControl;
}

// used by the Replay Controller to update which game is
// current
public void setGame(Game g) {

    this.game = g;
    game.refreshControlPanel();
    game.getDisc().update();
    for (int i=0; i<7; i++) {

        game.getRedTeam().getPlayer(i).update();
        game.getBlueTeam().getPlayer(i).update();
    }
}

public void start() {

    if ((gameThread == null) || (!gameThread.isAlive())) {
        gameThread = new Thread(this, "Game");
        gameThread.start();
    }
}

public void run() {

    while (true) {

        try {

            gameThread.sleep((int) TIME_PULSE);
        } catch (InterruptedException e) {}

        game.runGame(TIME_PULSE / 1000D);

        if (game.isRunning()) {
```

```
                recordTimer += (TIME_PULSE / 1000D);
                if (recordTimer >= 1D) {

                    replayController.recordGame();
                    recordTimer = 0D;
                }
            }
        }
    }
}
```

UltimateApplet.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.net.*;

//////////////////////////////////////////////////////////////////////
//
//  UltimateApplet.java
//
//  Author: Dean Bolton - 2002
//
//    Sets up the applet and calls the Ultimate.java class
//
//////////////////////////////////////////////////////////////////////


public class UltimateApplet extends JApplet  {

    public void init() {
        Ultimate uf = new Ultimate(getCodeBase(), true);
        setContentPane(uf);

        JPopupMenu.setDefaultLightWeightPopupEnabled(false);
        setJMenuBar(uf.getMenu());
    }
}
```

UltimateApplication.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;

//////////////////////////////////////////////////////////////////////
//
//  UltimateApplication.java
//
//  Author: Dean Bolton - 2002
//
//    Sets up the application and calls the Ultimate.java class
//
//////////////////////////////////////////////////////////////////////


public class UltimateApplication extends JFrame {

    public UltimateApplication() {
        super("Ultimate Simulator");

        setSize(new Dimension(500, 560));
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Ultimate uf = new Ultimate(null, false);
        setContentPane(uf);

        JPopupMenu.setDefaultLightWeightPopupEnabled(false);
        setJMenuBar(uf.getMenu());

        setVisible(true);

       uf.getFieldControl().setVisible(true);
        uf.getSimulationControl().setVisible(true);
    }

    public static void main(String[] args) {
       UltimateApplication frisbee = new UltimateApplication();
    }
}
```

World3D.java

```
package ultimate.simulator;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.behaviors.vp.*;

////////////////////////////////////////////////////////////////////////
//
//  World3D.java
//
//  Author: Dean Bolton - 2002
//
//    Class that is responsible for creating a displaying the
//    three-dimensional world
//
////////////////////////////////////////////////////////////////////////


public class World3D {

    private TransformGroup transformField = null;
    private Game game = null;
    private Canvas3D canvas3D = null;

    public SimpleUniverse simpleU = null;
    public BranchGroup world = null;
    public Transform3D lightTransform;
    public TransformGroup lightTransformGroup;

    public World3D(Game game) {

        GraphicsConfiguration config =
            SimpleUniverse.getPreferredConfiguration();
        canvas3D = new Canvas3D(config);

        world = new BranchGroup();

        Field3D field = new Field3D();
        transformField = field.getFieldGraph();

        transformField.addChild(game.getDisc().getTransformGroup());
        for (int i=0; i<7; i++) {

            transformField.addChild(game.getRedTeam().getPlayer(i).getTransformGroup());
            transformField.addChild(game.getBlueTeam().getPlayer(i).getTransformGroup());
        }

        setupLighting();

        world.addChild(transformField);

        world.compile();


        //SimpleUniverse is a convenient utility class
        simpleU = new SimpleUniverse(canvas3D);

        //experimental, seems to work well
        Transform3D zoomOutTransform = new Transform3D();
        zoomOutTransform.set(new Vector3d(0.0f,0.0f,3.0f));

simpleU.getViewingPlatform().getViewPlatformTransform().setTransform(zoomOutTransform);
        //
```

```
        OrbitBehavior orbit = new OrbitBehavior(canvas3D, OrbitBehavior.REVERSE_ALL);
        BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0),100.0);
        orbit.setSchedulingBounds(bounds);
        orbit.setZoomFactor(0.4);
        simpleU.getViewingPlatform().setViewPlatformBehavior(orbit);

        simpleU.addBranchGraph(world);

    }

    public Canvas3D getCanvas3D() {

       return canvas3D;
    }

    public void setupLighting(){

      Color3f aColor     = new Color3f(0.2f, 0.2f, 0.2f);
      Color3f lightColor = new Color3f(1.0f, 1.0f, 1.0f);

      // Create bounds for the lights
      BoundingSphere bounds =  new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0f);

      // Create the transform group node for the lights
      lightTransform = new Transform3D();

      Vector3d lightPos1 =  new Vector3d(0.0, 0.0, 2.0);

      lightTransform.set(lightPos1);

      lightTransformGroup = new TransformGroup(lightTransform);

      transformField.addChild(lightTransformGroup);

      // Create lights
      AmbientLight ambLight = new AmbientLight(aColor);
      Light        dirLight1;

      Vector3f lightDir1 = new Vector3f(lightPos1);

      lightDir1.negate();

      dirLight1 = new DirectionalLight(lightColor, lightDir1);

      // Set the influencing bounds
      ambLight.setInfluencingBounds(bounds);
      dirLight1.setInfluencingBounds(bounds);

      // Add the lights into the scene graph
      transformField.addChild(ambLight);
      transformField.addChild(dirLight1);

    }

    public void setUniverseZoom(float z){

        System.out.println("setUniverseZoom called. z: "+z);

        Transform3D oldTransform = new Transform3D();

        Transform3D newTransform = new Transform3D();


simpleU.getViewingPlatform().getViewPlatformTransform().getTransform(oldTransform);

        newTransform.set(new Vector3d(0.0,0.0,z));

        oldTransform.mul(newTransform);


simpleU.getViewingPlatform().getViewPlatformTransform().setTransform(oldTransform);
```

```
    }
    public void setUniverseRotationZ(double rot){
        System.out.println("inside setUniverseRotationX");
        Transform3D newTransform = new Transform3D();
        newTransform.rotZ(Math.PI / 4);
        Transform3D oldTransform = new Transform3D();
simpleU.getViewingPlatform().getViewPlatformTransform().getTransform(oldTransform);
        oldTransform.mul(newTransform);
simpleU.getViewingPlatform().getViewPlatformTransform().setTransform(oldTransform);
    }
    public void setSideView(){
        Transform3D translateTransform = new Transform3D();
        translateTransform.set(new Vector3d((Globals.BASE_FIELD_WIDTH /
                                    Globals.GRAPHICS_3D_CONVERSION),
                                    0.0, -2.858));
        Transform3D rotateTransform = new Transform3D();
        rotateTransform.rotZ(Math.PI / 2);
        translateTransform.mul(rotateTransform);
        Transform3D rotateTransform2 = new Transform3D();
        rotateTransform2.rotX(Math.PI / 2);
        translateTransform.mul(rotateTransform2);
simpleU.getViewingPlatform().getViewPlatformTransform().setTransform(translateTransform);
    }
    public void setCameraView(DoublePoint cp, double zHeading, double xHeading){
        Transform3D translateTransform = new Transform3D();
        translateTransform.set(new Vector3d(cp.getXLocation(),
                                    cp.getYLocation(),
                                    cp.getZLocation()));
        Transform3D rotateTransform = new Transform3D();
        rotateTransform.rotZ(zHeading);
        translateTransform.mul(rotateTransform);
        Transform3D rotateTransform2 = new Transform3D();
        rotateTransform2.rotX(xHeading);
        translateTransform.mul(rotateTransform2);
simpleU.getViewingPlatform().getViewPlatformTransform().setTransform(translateTransform);
```

```
    }

    public void setWorldTransform(Transform3D transform){
        transformField.setTransform(transform);

    }

}
```