Development Guide for UIST 2001 UI Design Contest

Authors: Marty Frenzel, Kathy Ryall, Joe Marks
Last updated: 4/05/01

CONTEST OVERVIEW

Objective

UIST 2001 will feature the first UIST Interface Design Contest. Contestants will have an opportunity to design and implement an interface to a real-time game application prior to the symposium. The game has been designed to accommodate a wide range of possible interfaces. During the competition contestants will use their interfaces to play the game against other contestants in a tournament. Prizes worth an estimated $2500 will be awarded to the winners. The goal of the contest is to encourage participants to explore interface software and technology in an applied setting, and to provide an opportunity for participants to showcase their work to the UIST community in an exciting and entertaining format.

Location

The contest will be held during the opening reception of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST 2001), which will be held in Orlando, Florida from November 11-14, 2001. For more information on the conference please visit the UIST web site, http://www.acm.org/uist.

Tournament

The goal of the design contest is to design the best interface you can for the application described below, and then to use your interface in competition with other design teams who will use their own interfaces. Participants are encouraged to develop original UI techniques and/or devices to gain a competitive advantage. Novel visualizations of the game situation; pen-based input techniques; two-handed input techniques; PDA-based multi-user interfaces; incorporating various forms of artificial intelligence into the interface: these ideas and more are all appropriate for inclusion in your UI design. The only requirement is that your UI communicate with the game application through the API described below.

At each round in the tournament winners will be determined by their game scores. The highest-scoring participants in a given round will advance to the next round. Note that for each round in the competition the game becomes progressively more difficult, as explained below. The ultimate winner will be the last group not eliminated. Additional prizes will be awarded to the best single-user UI and to the best UI designed by an all-student team.

How to Enter

To reserve a place in the contest, contestants must submit an entry form to the Conference Chair no later than Friday, October 12, 2001. Entry forms will be available online, starting in September. An Ethernet LAN will be provided for contestants to connect to the game server. Contestants are required to be present at the conference and to provide all hardware and software required for running their interface and for connecting it to the LAN hub.

GAME OVERVIEW

Objective

The game is played on a rectangular field populated with several obstacles (see Figure 1). The size, number, and location of obstacles may vary between rounds of the competition. A contestant will start with several players located arbitrarily on the board. The objective is to move as many of these players as possible to the finish line. The starting location and maximum speed of the contestant's players will vary between rounds of the competition. The contestant will start each round with a total of five players. The computer opponent starts the game with several players located arbitrarily on the board. The number, location, and maximum speed of the opponent's players will vary between levels of the competition. The opponent's players move toward the competitor's players whenever they have an unobstructed view of them; otherwise the opponent's players move randomly. The velocities of the contestant's players are controlled via his/her user interface. If one of the opponent's players contacts any of the contestant's players, the contestant's player is captured and can no longer move. A game ends when either all the contestant's players have been caught or crossed the finish line, or when time has expired.

Scoring

A contestant's score is the number of players that successfully reach the finish line. In the event of a tie, the time at which the contestant's last player crossed the finish line will be used to determine the winner, with an earlier time beating a later time.

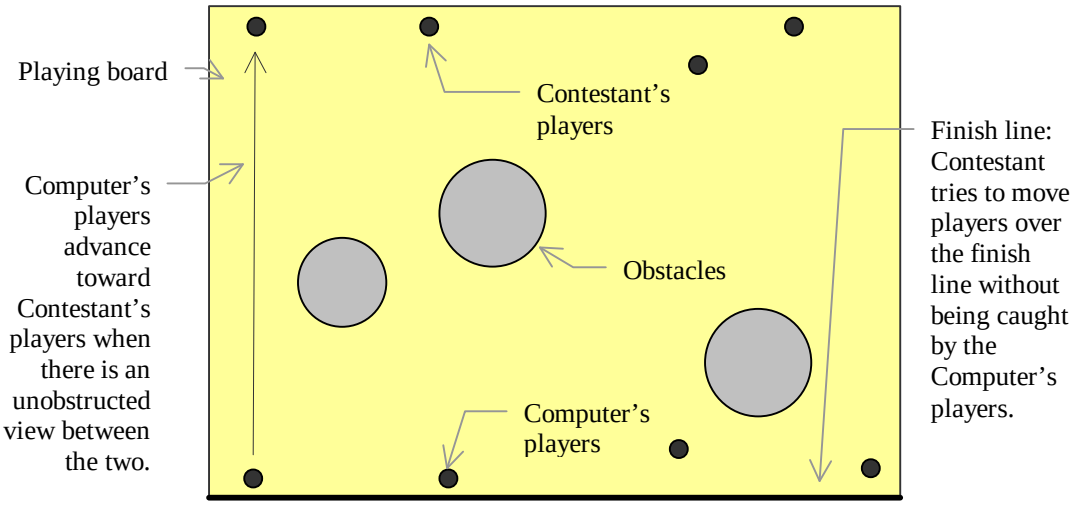

Figure 1: The Playing Field

Designing the Interface

There are no constraints on the design of your UI, except that it must communicate with the game application through the API provided. We have provided a reference interface to illustrate the use of the API. While our implementation is in Java and runs on a standard PC or Unix computer, your interface may be implemented in a language of your choosing, running on any hardware you would like to use. The game is structured as a client-server application. Client-server communication is conducted via the Common Object Request Broker Architecture (CORBA), although you do not need to know much about CORBA to make use of the Java API that we provide.[1] While the tournament will be run in a distributed environment over a local-area network, the game is structured so that development may be done on either a single computer or in a distributed environment.

[1] If you use another language, you will need to learn a little about CORBA – see the next section.

GAME API

UML Class Diagram

An API is provided for the game and contestants are required to program to that model. A UML-class design diagram of the game is provided below. Of particular interest are the methods provided by the GameBrain interface. These methods define how communication happens with the game server. Data types for properties are specified as Interface Design Language (IDL) data types.

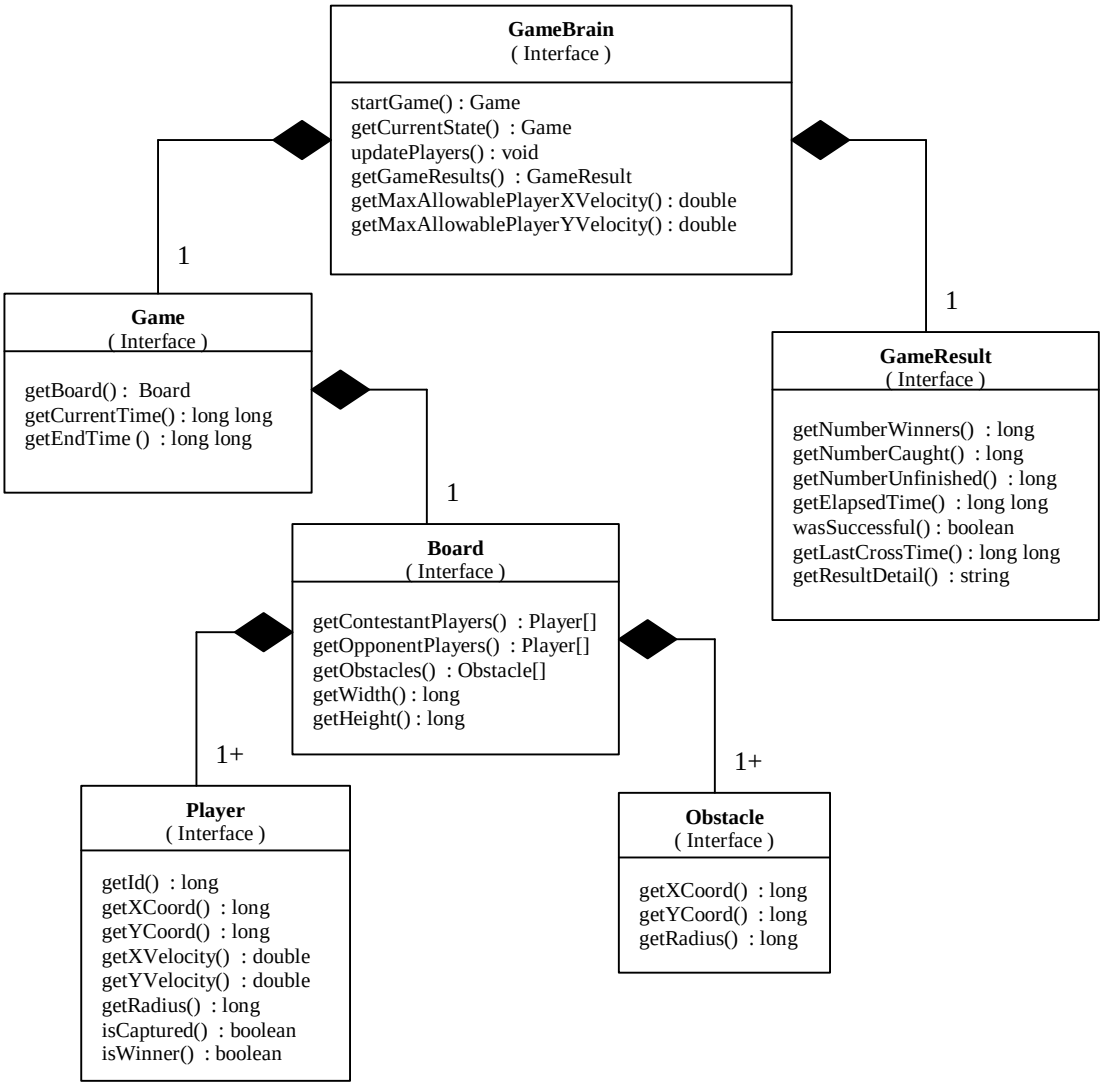

Figure 2: UML Class Diagram of Game Application

CORBA Wrapper

Client-server communication is provided by the CORBA protocol. We have provided Java wrappers to CORBA, so no familiarity with CORBA is required if you use Java to implement your interface. People who want to use another programming language will need some familiarity with CORBA in order to include its functionality in your program. For more information on CORBA see:

- http://www.omg.org – homepage for the Object Management Group, the designers and maintainers of CORBA.
- http://www.omg.org/gettingstarted/corbafaq.htm: the CORBA FAQ.

The IDL file for the game can be found in Appendix A of this document. In addition to providing a more detailed view of the classes, methods, and exceptions involved with the game, the IDL file should be used as the starting point for your IDL-to-programming-language conversion. If your UI will be in a language other than Java, you will need to perform this conversion. However, if your UI is in Java, you can just import the classes included with the reference implementation UI.

EXPLANATION OF REFERENCE USER INTERFACE

The server side of the game has been provided to you along with a reference implementation UI that should be used to familiarize yourself with the game. These contents are packaged in a zip file that may be downloaded and installed using the following instructions.

Server Requirements

- Any Java-compliant operating system
- JDK1.3
- 128 MB RAM
- Pentium II 350 MHz processor

Client Requirements

- Any Java-compliant operating system
- JDK1.3
- 64 MB RAM
- Pentium II 350 MHz processor
- Can be the same machine as the server for development purposes

Installation

1. Download zip file from http://www.acm.org/uist/contest01.
2. Unzip the file into a temporary directory. This step should create the following file structure:

UISTgame/	
/startServer.bat	starts CORBA naming service and game server on MS Windows
/startDisplay.bat	starts game UI on MS Windows
/startServer.sh	starts CORBA naming service and game server on UNIX
/startDisplay.sh	starts game UI on UNIX
/game_1.0.idl	IDL file with class definitions and full method signatures
/lib/classes.jar	jar file containing required game classes
/lib/xml.jar	jar file containing required xml classes for game setup
/media/caught.wav	sound file that is played when a player is caught
/xml/game_1.0.dtd	DTD file used to validate syntax of setup XML files
/xml/setup.xml	a sample XML setup file to define the start state of the game
/src/game/Display.java	source code for the Java UI reference implementation

Running the Server and Interface

The client and server can be on the same or on different Windows machines. Running the client and server under Unix is similar.

1. Open a command prompt and cd into the `UISTgame` directory.
2. If working on UNIX, set execute permissions on the scripts by running this command:
   ```
   chmod +x *.sh
   ```
3. Start the CORBA naming service and the server with this command:
   ```
   .\startServer.bat   (./startServer.sh on UNIX)
   ```
4. Start the Display with the command:
   ```
   .\startDisplay.bat  (./startDisplay.sh on UNIX)
   ```

Alternatively, you could ignore the scripts and type the full commands**:**

1. Open a command prompt and cd into the `UISTgame` directory.

Running the Server and Interface (cont'd)

2. Start the CORBA naming service with the command:
   `tnameserv –ORBInitialPort <port>`
3. At a separate command prompt, cd into the `UISTgame` directory and start the game server with the command (all on one line):
   ```
   java -classpath .\lib\classes.jar;.\lib\xml.jar;%classpath%
   game.GameSetup -xmlFile .\xml\setup.xml [–ORBInitialPort <port> ]
   [-ORBInitialHost <host> ]
   ```
4. At a third command prompt, cd into the `UISTgame` directory and start the game Display with the command (all on one line):
   ```
   java -classpath .\lib\classes.jar;%classpath%
   game.Display [–ORBInitialPort <port> ] [-ORBInitialHost <host> ]
   ```

Helpful tips:

- If running on a UNIX machine, the classpath specification must be in UNIX format as shown below
  `-classpath ./lib/classes.jar:./lib/xml.jar:$CLASSPATH`
- quotation marks, eg. `"%classpath%"`, may be required if your classpath contains directories with spaces in the name
- the `tnameserv` utility can be found in the <jdk1.3>/bin directory
- the naming service, game server, and game display can all be run on different machines. To accomplish this, use the `–ORBInitialHost` and `–ORBInitialPort` arguments to specify the machine and port, respectively, that the naming service is running on. By default, the naming service starts on port 9999. This port requires root user access on UNIX. Therefore, if the naming service is run on a UNIX machine, you may be required to specify the `–ORBInitialPort` argument.

Playing the Game

At startup, a brief description of the game is presented. Click on the button to start the game. Contestant players (pastel colors) are aligned at the top of the screen and opponent (computer) players are the blue circles at the bottom of the screen. You can use the up, down, left, and right arrows on your keyboard to control the velocities of the contestant players so as to move them over the cyan line at the bottom of the screen. There are five contestant players to control. To control player 1, press the '1' key on the keyboard and then use the arrow keys; to control player 2, press the '2' key on the keyboard and then use the arrow keys; and so on for the other players. Of course this is not a very usable interface, but it can serve as a starting point for the development of your own interface.

Defining XML Files

The starting configuration for a game is defined by the XML file located at `UISTgame/xml/setup.xml` . You are encouraged to play with this XML file to experiment with new game configurations and vary settings such as the game duration, maximum velocities of players, and number of opponents / obstacles. However, there are several restrictions to be aware of when modifying the file:

- The file must list exactly 5 <contestant> tags.
- The file can list no more than 10 and no less than 1 <opponent> tags.
- The file can list no more than 10 and no less than 1 <obstacle> tags.
- The file must be well formed with respect to the xml/game_1.0.dtd file.

In order for the application to pick up changes to the XML file, you must stop and restart the GameSetup class (startServer script, or step 3 from above).
IMPORTANT: at the UIST competition, configuration settings will be controlled by game administrators and not by contestants. Accordingly, user interfaces should be designed to accommodate many different game configurations.

Common Problems

- The game ends prematurely because 'Deviation exceeds the allowable.'
  - The server automatically stops a game if the elapsed time between updates of player positions exceeds the specified maximum elapsed time. NOTE: this error is more likely to occur on slower machines and machines with insufficient RAM.
  - Suggested solution: use a machine with a faster CPU and/or allocate more RAM to the java server process with the '–Xmx' argument. For example, use the command:

```
java –Xmx192m -classpath ./lib/classes.jar;./lib/xml.jar;%classpath%
   game.GameSetup -xmlFile ./xml/setup.xml [–ORBInitialPort <port> ]
[-ORBInitialHost <host> ]
```

- The display fails with 'COMM_FAILURE' trying to start the game.
  - The display can not contact the game server.
  - Suggested solution: check the status of the game server.

- The naming service (`tnameserv`) does not start on UNIX.
  - By default, the naming service starts on port 9999. This port requires root user access on UNIX. Therefore, if you do not have root priviledges, you must specify the `–ORBInitialPort` argument. For example,

```
tnameserv –ORBInitialPort 27790 &
```

APPENDIX A: IDL DEFINITION FILE FOR GAME APPLICATION

```
// ========================
// This file contains the IDL specification for the game that is to
// be part of a user interface design competition at the UIST '01
// conference in Orlando, Florida.
//
// The objective of the game is for a contestant to move their players
// across a playing field without being caught by the opponent.
//
// Author: Marty Frenzel
// Version: 1.3
// Date: April 26, 2001
// ========================

module game {

        // define the Player interface
        // -----------------------
        interface Player
        {
                // returns the unique id number of this player
                long getId();

                // returns the X coordinate of the center of this player
                long getXCoord();

                // returns the Y coordinate of the center of this player
                long getYCoord();

                // returns the X velocity of this player
                double getXVelocity();

                // returns the Y velocity of this player
                double getYVelocity();

                // returns the radius (in pixels) of this player
                long getRadius();

                // returns true if this player has been captured
                boolean isCaptured();

                // returns true if this player has crossed the finish line
                boolean isWinner();
        };

        // define an array of Players
        // ------------------------------------------
        typedef sequence<Player> PlayerArray;
```

```
 // define the Obstacle interface
 // -------------------------
 interface Obstacle
 {
        // returns the X coordinate of the center of the obstacle
        long getXCoord();

        // returns the Y coordinate of the center of the obstacle
        long getYCoord();

        // returns the radius (in pixels) of this obstacle
        long getRadius();
 };

// define an array of Obstacles
// ------------------------------------------------
typedef sequence<Obstacle> ObstacleArray;

// define the Board interface
// the Board class holds a complete snapshot of all Players and all obstacles; as well as the
// height and width of the board
// ------------------------------------------------
interface Board
{
   // the Players that are controlled by the Contestant
   PlayerArray getContestantPlayers();

   // the Players that are controlled by the Computer
   PlayerArray getOpponentPlayers();

   // the Obstacles that exist
   ObstacleArray getObstacles();

   // returns the width of the board
   long getWidth();

   // returns the height of the board
   long getHeight();
 };

 // define the Game interface
 // the Game class holds a Board; as well as a timestamp indicating the current time and a
 // timestamp indicating the time at which the game will end.  Both timestamps are
 // specified as times on the Server machine
 // ------------------------------------------------
 interface Game {
        // returns the Board associated with this game
        Board getBoard();

        // timestamp (milliseconds) indicating the current time
        // on the server when GameBrain.getCurrentState() was called
        long long getCurrentTime();

        // timestamp (milliseconds) indicating time on server when game will end
        long long getEndTime();
   };
```

```
        // define the GameResult interface
        // the GameResult holds information about the outcome of a game
        // -----------------------------------------------
        interface GameResult
        {
                // number of players who crossed the finish
                long getNumberWinners();

                // number of players who were caught by opponent
                long getNumberCaught();

                // number of players who didn't cross the finish and
                // weren't caught
                long getNumberUnfinished();

                // elapsed amount of time the game took (milliseconds)
                long long getElapsedTime();

                // indicates whether game finished properly or an error occured
                boolean wasSuccessful();

                // if the game terminated unsuccessfully, this is a description
                // of what happened
                string getResultDetail();

                // time at which the last player to cross the finish line did so
                long long getLastCrossTime();
        };

  // define the exception thrown if the Contestant attempts to
  // update settings for an invalid Player
  // -----------------------------------------------
  exception InvalidPlayerException{
        string mMessage;
  };

// define the Exception thrown if the contestant tries to set
// a Player's velocity outside of the allowable velocity range
// -----------------------------------------------
exception InvalidVelocityException{
        string mMessage;
  };

 // define the exception that is thrown if a method is called which activates
 // activity in the server that should not occur while a game is running
 // -----------------------------------------------
exception GameRunningException{};

  // an exception that is thrown if the game is not running and someone calls
  // getCurrentState() on the GameBrain
  // -----------------------------------------------
  exception GameNotRunningException{
        string mMessage;
  };
```

```
    // define the Brain that controls the game and monitors
    // Player's locations, etc.
    // user interfaces will lookup a GameBrain in the CORBA naming service and then
        // use that brain to interact with the game.
        // ----------------------------------------------------
        interface GameBrain{

            // used to obtain a current snapshot of the Game
            Game getCurrentState()
                raises (GameNotRunningException);

            // used to update the state of a Contestant's Player
            // throws InvalidPlayerException if the specified PlayerId does not correlate to a
            // valid Player.
            // throws InvalidVelocityException if the specified exception is
            // outside the range of the allowable velocities.
            // NOTE: the range of allowable velocities may vary from game to game.
            void updatePlayer( in long mPlayerId, in double mNewXVelocity, in double mNewYVelocity )
                raises (InvalidPlayerException, InvalidVelocityException, GameNotRunningException);

            // returns the current max allowable Player velocity in the X direction
            double getMaxAllowablePlayerXVelocity()
                raises (GameNotRunningException);

            // returns the current max allowable Player velocity in the Y direction
            double getMaxAllowablePlayerYVelocity()
                raises (GameNotRunningException);

            // returns the current max allowable opponent velocity in the X direction
            double getMaxAllowableOpponentXVelocity()
                raises (GameNotRunningException);

            // returns the current max allowable opponent velocity in the Y direction
            double getMaxAllowableOpponentYVelocity()
                raises (GameNotRunningException);

            // used to start the Game
            Game startGame()
                raises (GameRunningException);

            // used to retrieve the results of the last game that was run
            // this method is only valid between games
            GameResult getGameResults()
                raises (GameRunningException);
        };

};   // end of module game
```