Robot Gladiators: A Java Exercise with Artificial Intelligence

David S. Yuen & Lowell A. Carmony

Department of Mathematics & Computer Science
Lake Forest College
555 N. Sheridan Road
Lake Forest, IL 60045
Yuen@LFC.Edu & Carmony@LFC.Edu

ABSTRACT

This paper describes an interesting game, Robot Gladiators, and the Java program that permits students to program the strategy of individual robots. As such, the program is appropriate for a first Java class or a first class in Artificial Intelligence.

Robot Gladiators is a multiplayer game in which the goal is to have your robot survive against all other robots. The other robots can represent other people's strategies or they can be "dumb" robots supplied by the program. At the start of the game, each robot is placed randomly on a rectangular playing field. Each robot can turn, move, get its current location, check its long-range directional radar or its short-range proximity radar, or fire its cannon. The action of each robot is controlled by an individual run() method. Each student programs his/her own robot and then, by letting the game play for 100 trials, one can begin to judge the soundness of different strategies.

The paper describes the main Java program that controls the actions of the robots. More precisely, the main program referees the actions of the robots, which are requested from individual threads associated to each robot. The ease of implementing multi-threaded programming in Java makes it an ideal language to use [1]. In particular, the interplay of a graphics thread, a game thread, and the individual robot threads will be explained. The program also collects statistics over many games, making it possible to play a series of games. The Java code will be made available to anyone who wants it. In a Java class, this code can be used as an example of a moderately sized Java project.

On the other hand, the Java code for each robot is very simple. Even non-Java students can create their own robot code by merely imitating the class definition of the sample dumb robot. Basically, the student devises a strategy and then implements that strategy by defining a new subclass of the specified Robot class, which invokes the appropriate methods (move(), turn(), fire(), etc.) from this superclass Robot. Without having to be concerned about the graphics and threads, this program can be used in an introductory AI class or in a class where one wants a good example of object-oriented programming without getting heavily involved in Java.

The paper presents a learning situation that is very open ended for the student. We have found that students get very excited about Robot Gladiators and want to continue to implement

improved strategies when they see how their current strategies perform. We have run the program with as many as six student robots and four dumb robots and have found that the program's performance is good with up to ten robots at a time.

THE GAME OF ROBOT GLADIATORS

In Robot Gladiators, individual human programmers pre-program their robots to act on its own in an environment with other robots. The goal of the game is to destroy all other robots. The action occurs in a rectangular arena of default size 400 by 200 pixels, but this size can easily be adjusted by the user. In the arena, a robot appears as a circular disk of default radius 10 pixels. A robot always faces a certain direction, denoted graphically by a triangular arrowhead. Figure 1 shows what Robot "a" looks like on the screen.

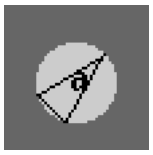

Figure 1

Each robot is capable of the following actions:

- A robot may rotate itself. The default rotational speed is 22 degrees per second.

- A robot may move forward in the direction it is facing. The default speed is 30 pixels per second. A robot stops moving forward if it encounters an obstacle, such as a wall or another robot.

- A robot may fire one shot at a time. It takes 0.1 seconds to ready and fire the shot. The shot appears on the screen as a small disk of radius 3 pixels and the shot travels at 50 pixels per second. The robot may not fire again until the fired shot has exploded, either by hitting another robot or by leaving the boundaries of the arena.

- A robot may inquire whether it is capable of firing a shot at the present time. There is no time penalty for this action.

- A robot may use its proximity radar detector, which returns the number of robots within 100 pixels. This action takes 0.1 seconds to execute.

- A robot may use its forward radar, which returns the number of robots within 30 degrees of its forward direction and within a radius of 200 pixels. This action takes 0.1 seconds to execute.

- A robot may obtain its present position and present heading. These actions do not cost time.

- A robot may also obtain the dimensions of the arena. This action has no time cost.

A robot is destroyed if hit by any shot. The last robot to survive is declared the winner. At the beginning of each battle, all robots are randomly placed in the arena in a non overlapping fashion. There is a built-in delay of 5 seconds before robots may fire any shots at each other.

Clearly, the default values used above are constants and are merely suggested as starting values. They may be altered by the user.

THE OUTLINE OF THE PROGRAM

We now describe the program that controls the robots. Throughout this paper, we will use Java, but it is possible to use any object-oriented language which supports multi-threading. There are two classes **RobotMaster** and **Robot**, which control the battle, and individual human programmers write their own subclass of Robot to control their own robots. We shall refer to these subclasses as the **RobotGladiator** classes.
The RobotMaster class is the main class, which referees all the action of the robots and draws the graphics. The game data is stored in various arrays, indexed by the robots, that store the position and heading and status (such as alive, radar on, etc.) of each robot and their shots and other control variables.

At the heart of the action are two threads, the graphics thread and the game thread. The graphics thread merely draws the graphics literally according to the data arrays (at a default refresh rate of 20 frames per second). The game thread moves and updates the shots and detects whether any robots have been hit and takes corresponding actions.

The Robot class defines methods for communicating with the RobotMaster object. These methods are called by the individual RobotGladiator objects, and include such methods as move(), turn(), fire(), etc. The details of these interactions are described in the next section.

The RobotMaster object generates a window with the arena, along with various Graphics User Interface controls and displays. The records of each robot are displayed, such as rank in the current battle, average rank, number of wins, and number kills in all battles so far. The GUI controls include:

- a choice menu to select the number of additional dumb robots
- a choice menu to select the number of trials
- a start button to start a series of trials,
- an abort button to abort the current trial
- an abort button to abort all trials
- a pause and resume button that toggles
- a sound on and sound off button that toggles
- a game speed up or slow down scroll bar
- and a graphics refresh rate scroll bar.

Also, since each trial is limited to a fixed time (the default is 10 minutes),
the amount of time remaining in the current trial is also displayed. Figure 2 shows a screen shot of the program in action.

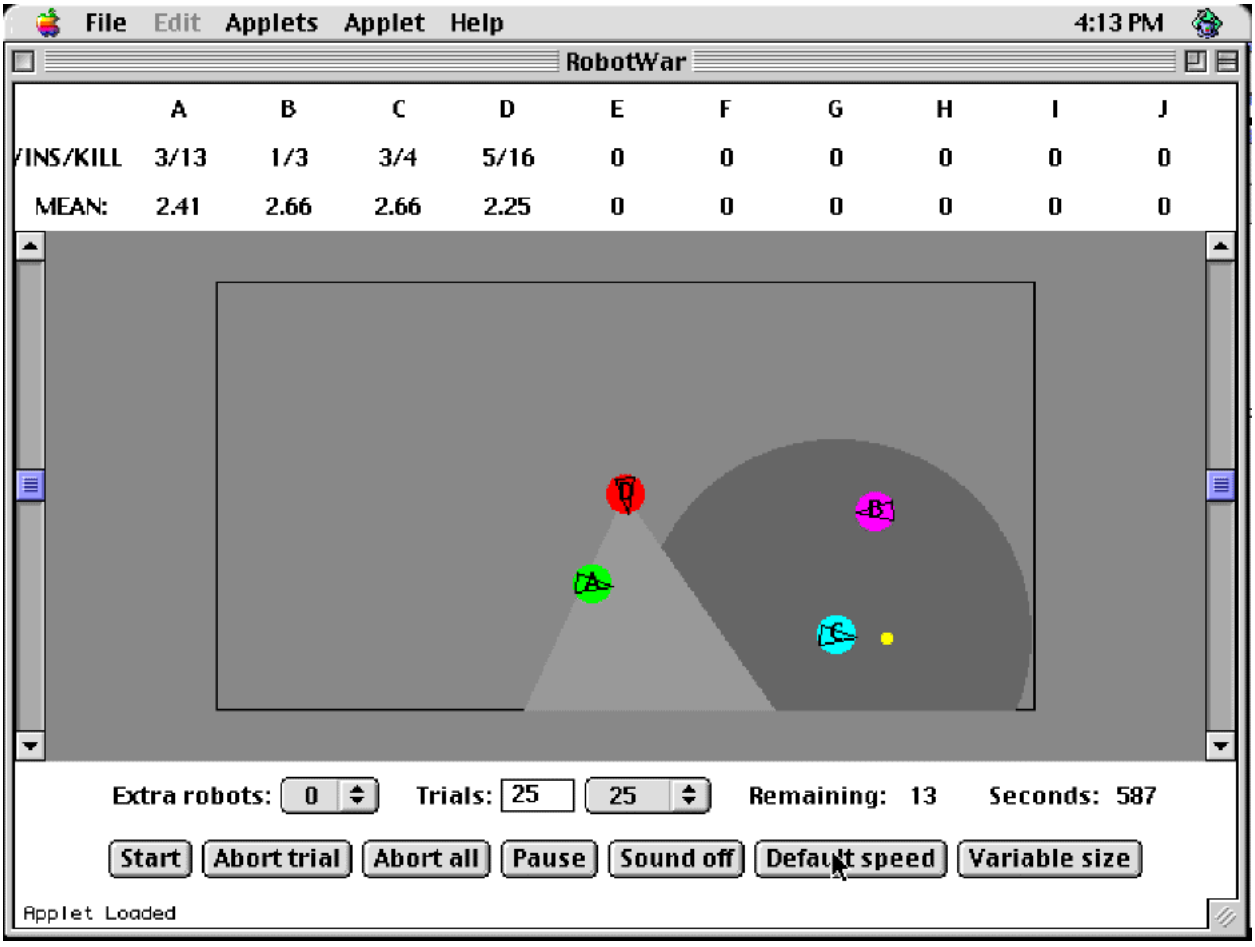

Figure 2

In Figure 2, note that Robot C, in the lower right corner, has just fired a shot (at nothing). Robot C also has its proximity detector on, as is indicated by the gray circle around Robot C. Note that Robot B would be detected by this proximity detection. Further, note that robot D, in the center of the arena, has its forward radar on, as is indicated by the lighter cone emanating from Robot D. Robot A would be detected by this radar. Appropriate sounds are played when either kind of detection occurs.

THE INTERACTION BETWEEN THE RobotMaster AND Robot CLASSES

At the beginning of a battle, an array of Robot objects is created, one for each contestant using the corresponding subclasses. A parallel array of Thread objects is also created using each of these Robot objects as targets.
Each thread is then started, which corresponds to the run() method in each RobotGladiator class being called.

As an example, here is the Robot method fire().

```
public class Robot implements Runnable
{
  boolean fire()
  {
    return RobotMaster.fire(this); //Pass Robot object itself
  }

  ...
}

public class RobotMaster extends Applet implements Runnable
{
  static Robot[] robotObjects;
  static double[] x, y, dir, shotX, shotY, shotDir;
  static boolean[] shotOn;
  static int numberOfRobots, robotRadius, shotRadius;
  static double FIREDELAY=100, gameSpeedFactor=1;
  ...

  static int getIndexOfRobot(Robot r)
  {
    int i;
    for(i=0;i<numberOfRobots;i++)
    {
      if(robotObject[i]==r)return i;
    }
    return -1; //and or report error!!
  }

  public static boolean fire(Robot r)
  {
    try{Thread.sleep((int)(FIREDELAY*gameSpeedFactor);}
    catch(InterruptedException e){}   //Put Robot to sleep
    int n = getIndexOfRobot(r);
    if(shotOn[n])
    {
      return false;   //You already have an active shot.
    }
    else
    { //place new shot just in front of robot
      shotX[n] = x[n] +

(robotRadius+shotRadius)*Math.cos(dir[n]*3.14159/180);
      shotY[n] = y[n] +
```

```
(robotRadius+shotRadius)*Math.sin(dir[n]*3.14159/180);
      shotDir[n] = dir[n];  //shot's direction same as robot's.
      shotOn[n] = true;     //shot is active
      fired = true;         //turn control variable on;
      return true;
    }
  }
```

When the fire() method of Robot is called, the static method fire() of RobotMaster is called with the Robot object itself as the argument. A static method is used so that the Robot object need not have access to the RobotMaster object itself. This simplifies the programming slightly. The main idea is that by passing itself as an argument, the Robot object allows the RobotMaster object to determine which RobotGladiator is calling the method, and hence can update all appropriate variables. Notice that the RobotGladiator is put to sleep for an appropriate amount of time (FIREDELAY is the constant 100 milliseconds, and gameSpeedFactor can be adjusted by the game speed scrollbar to speed up or slow down the game). The control boolean variable "fired" is used in the game thread to play the corresponding sound.

We now give a full list of the Robot methods. These are the methods that are available to the human programmer of a robot.

```
boolean move(double distance)
//moves the robot forward by distance pixels;
//returns true if successful, false if obstacle encountered;
//default speed is 30 pixels per second.

void turn(double degrees)
//turns robot by specified degrees, positive or negative;
//default angular velocity is 22 degrees per second.

boolean fire()
//attempts to fire a shot; returns true if successful;
//takes 0.1 second to complete this action.

int proximity()
//returns the number of robots within proximity (100 pixels);
//takes 0.1 second to complete this action.

int radar()
//returns the number of robots within forward radar
//(within 30 degrees left and right, and within 200 pixels);
//takes 0.1 second to complete this action.

boolean fireAble()
//returns true if capable of firing. (no time charged)
```

```
double getX()
//returns x-coordinate of robot. (no time charged)

double getY()
//returns y-coordinate of robot. (no-time charged)

double getDirection()
//returns heading of robot in degrees. (no time charged)

double getWidth()
//returns arena width, maximum x-coordinate. (no time charged)

double getHeight()
//returns arena height, maximum y-coordinate. (no time
charged)

double getScore()
//returns robot's current average rank from previous battles.
//(no time charged)
```

For an advanced class of students with good math skills, the above mentioned basic methods should suffice.

For students at a more elementary level, one could also make more high-level methods available in the Robot class. These might include:

```
void turnTo(double direction)
//turns robot to specified heading by calculating
//appropriate number of degrees and calling turn();

void turnTowards(double x, double y)
//turns robot towards the point (x,y) by calculating
//the heading towards (x,y) and then calling turnTo().

moveTo(double x, double y)
//moves robot to specified coordinates by calling turnTo(x,y)
//followed by finding the distance to (x,y) and calling
move().

boolean fireIfAble()
//fires a shot if able by using fireAble() and then fire().

boolean snoozeTilFire()
//waits (by sleeping) until able to fire, then fires.
```

These additional methods are also good for students in a one-time lab experience, such as at a colloquium. They make it possible to quickly implement Robot strategies.

One annoying situation that can arise is when a student's RobotGladiator has an infinite loop in which it never goes to sleep. This can cause the entire battle to slow down significantly. The best thing to do in this situation is to abort all trials and inspect the code of each RobotGladiator. One could also overcome this problem by requiring all RobotGladiators to perform actions periodically. This could be monitored and if a robot didn't perform any actions in a given period, it would be declared "rusted" and would be destroyed. If this monitoring were done, one would also need to provide a sleep() method that the monitor recognized as a legal action.

THE MAIN RobotMaster CLASS IN JAVA PSEUDO-CODE

The complete Java code is too lengthy to include here, but will be made available to anyone who wants it. Here are some of the more important methods for the main RobotMaster class, written in Java pseudo-code.

```
public class RobotMaster extends Applet implements Runnable
{
  //Define all constants, arrays, variables.

  public void init()
  {
    Create GUI environment, initialize all variables.
  }

  public boolean action(Event e, Object arg) //For Java 1.0
  {
    Take corresponding action when certain buttons are pressed.
  }

  void startBattle()
  {
    Create a graphics thread and a game thread;
    Create all the robots; randomly place the robots;
    start all threads.
  }

  void runGraphicsThread()
  {
    Draw all robots and shots literally according to data
arrays;
    Sleep some milliseconds. Repeat forever.
  }

  void runGameThread()
  {
    Move and update all active shots. Detect any hits and take
    appropriate actions.  Check sound control variables and play
```

```
    appropriate sounds.  Sleep some milliseconds. Repeat until 1
    or 0 robots are left and no active shots remain.
  }

  All the methods called by the Robot class...

}
```

SOME RobotGladiator STRATEGIES

The following are some elementary strategies that our students have programmed for their robots.

- "Random": Just randomly move, turn, and fire.
- "The Corner Strategy": Go to the nearest corner, and turn back and forth and randomly fire within the 90 degree arc.
- "The Roaming Lighthouse": Go to some location, and turn in a circle until the forward radar detects something, in which case stop and fire. Go to a different random location after a period of time.
- "Spin in the Middle": Go to the center of the arena, spin and fire randomly.

To illustrate the possibilities and make the student's task easier, we supply the students with the code for a "Dumb" robot. Here is the code:

```
public class RobotDumb extends Robot
{
  public void run() //This will be called ONCE for EACH game in
                    //the set of trials.
  {
    double r;
    while(true)
    {
      r=Math.random();
      if (r<.25) //25% of the time fire, possibly at nothing
      {
        if (fireAble()) fire();
      }
      else if(r<.5) //25% of the time move forward
                     //randomly from 1 to 20 pixels
      {
        move(Math.random()*20+1);
      }
      else if(r<.8) //30% of the time turn
                    //randomly from -50 to 50 degrees
      {
```

```
        turn(Math.random()*100-50);
      }
      else if (r<.9) //10% of the time turn on
                     //proximity detector, ignoring results
      {
        proximity();
      }
      else //10% of the time turn on radar, ignoring results
      {
        radar();
      }
    }
  }
}
```

Dumb is not a very good robot, but it illustrates the correct syntax of the main methods that are available for programming robot strategies. In fact, students can create their own robot by simply modifying the code for the Dumb robot. This makes it possible to use RobotGladiators with students who are not proficient Java programmers or in short presentation, such as a colloquium.

FURTHER IDEAS

Students will suggest and, if they are capable, they can program additional methods for the robots. For example, one could allow the radar beam to be adjusted via a method such as

```
setRadarAngle(d ouble radarAngle)
```

One could also allow a defensive method that detects radar, such as

```
boolean radarDetected()
```

CONCLUSIONS

In an elementary class, this game allows a student to program a fun robot game while learning the concepts of classes and methods.

This game is also suitable as a one-time demonstration.

In an advanced programming class, the students would have to write their own high-level methods.

In an Artificial Intelligence class, the robots should have a more sophisticated search and destroy (and possibly duck) algorithm, and they should also learn from previous battles by using the getScore() method to assess their current strategies.

REFERENCES

1. CORE Web Programming, Marty Hall, Prentice Hall, 1998.