WU-BOTS - A ROBOT SIMULATOR FOR STUDENTS *

Bruce Mechtly, Joshua Wurtz and Tyler Wade
Department of Computer Information Sciences
Washburn University
Topeka, KS
bruce.mechtly@washburn.edu

ABSTRACT

We introduce the Java WU-Bots Robotic Simulator as an alternative to using robots like Scribbler™ and Finch™ in the classroom. Students can easily design a maze of walls using the environment editor, then program the robots using ordinary Java code in a simple text editor. Up to 10 robots can be programmed to run simultaneously. Projects and environments can be saved and loaded easily. We also present a set of projects to challenge students of all levels, including an autonomous robot that explores a maze and generates a graph representation which it can then use to navigate the graph using a shortest path algorithm.

INTRODUCTION

While there are many robotics simulators available [1], including many that are open-source [2], we chose to design and implement our own simulator as a student project. The result is a simple two-dimensional environment where up to 10 robots can interact with walls and each other. We have also created a three-dimensional environment using OpenGL, but have not yet fully integrated this feature into our program.

After using the Scribbler™ robot in the classroom for many years, we were frustrated with the need to constantly replace batteries. We purchased a set of rechargeable batteries, but found they would not last through even one class period. Also, there were often slight imperfections in the speeds of the motors which needed correction. While not a problem for advanced students, this was too difficult for introductory students.

Our goal was to produce a simple Java program that can run on virtually any computer that reliably simulated the experience a student would have with currently available classroom robots. The simulator needed to include an environment editor and a code editor, all integrated for easy use. The program is deployed as a Java executable jar file and is available here [3]. It includes all source code.

In what follows we will discuss the design and organization of the WU-Bots program, including the internal design which may be of interest to advanced students interested in graphics and animation. We then introduce a set of simple student projects that can be used to challenge students at all levels.

THE SIMULATOR

A screenshot of the simulator is shown in Figure 1. There are three tabs that allow the user to easily navigate between the different panels. The "Simulator" tab shows the environment that the robots explore. In this project there are five robots. Robots can leave "breadcrumbs" and "markers" behind to show their past motion. The "Robot Control" tab is used to go to the code editor and the "Commands" tab gives a summary of the method calls the student can use to control the robot.

Note the presence of the "Edit" button on the "Simulator" tab. This is a toggle button that takes the user to a graphical editor where they can add walls, change the position of walls, and change the initial position and orientation of the robots.

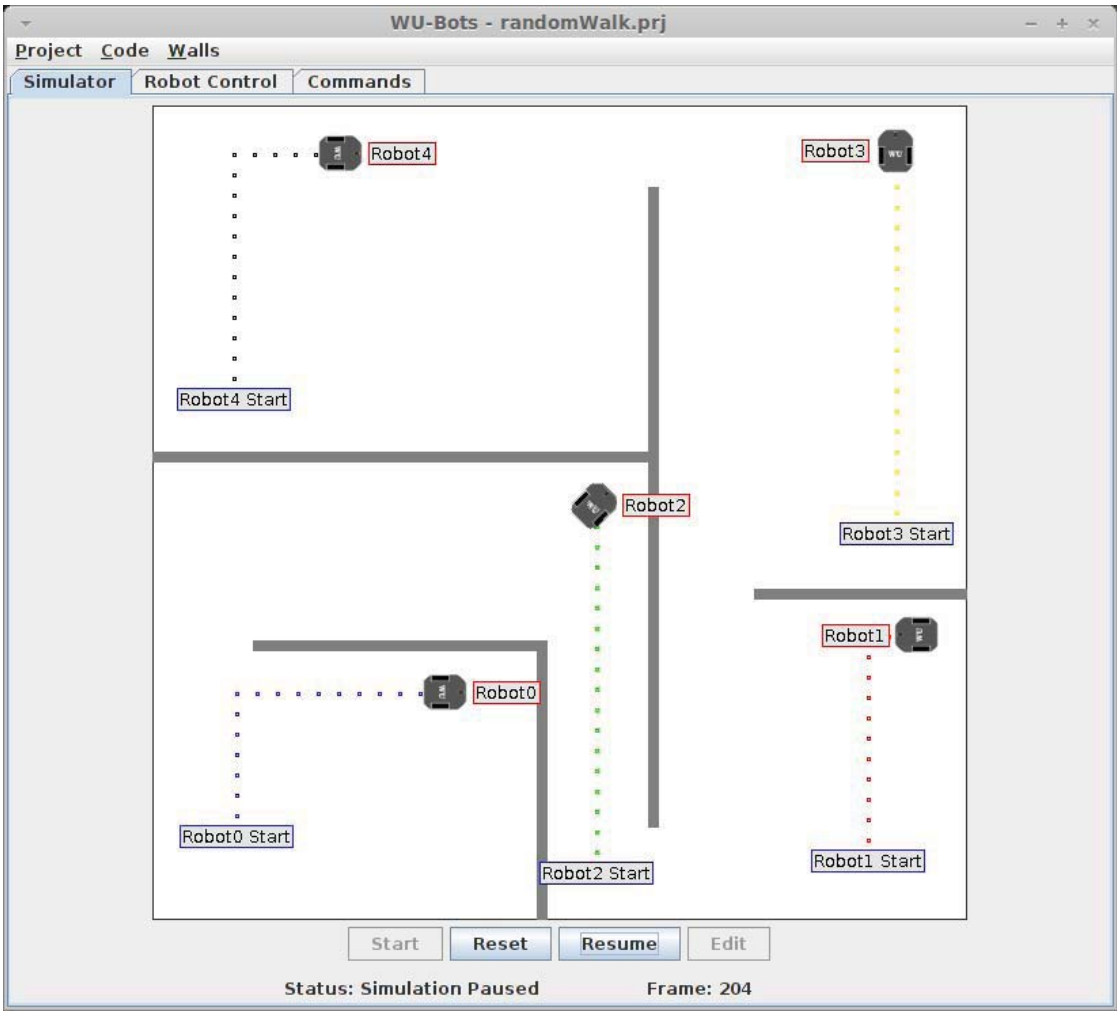

Figure 1. The Simulator Tab

Clicking on the "Start" button will compile all active robots (if not already compiled) and then begin the simulation. When the simulation is running one can pause and resume it whenever one chooses. Clicking on "Reset" will return the robots to their initial position. Robots will stall when they hit a wall or another robot due to collision detection within the simulator. The student can use the sensor to detect an object in front of the robot to avoid collisions.

Figure 2 shows the "Robot Control" tab of the simulator. The window on the left contains the source code and the window on the right contains complier messages. Note the spinner to select which robot is being edited. The name of the robot can be changed in the text box. The "Active" checkbox is used to tell the simulator which robots to show and which to hide.

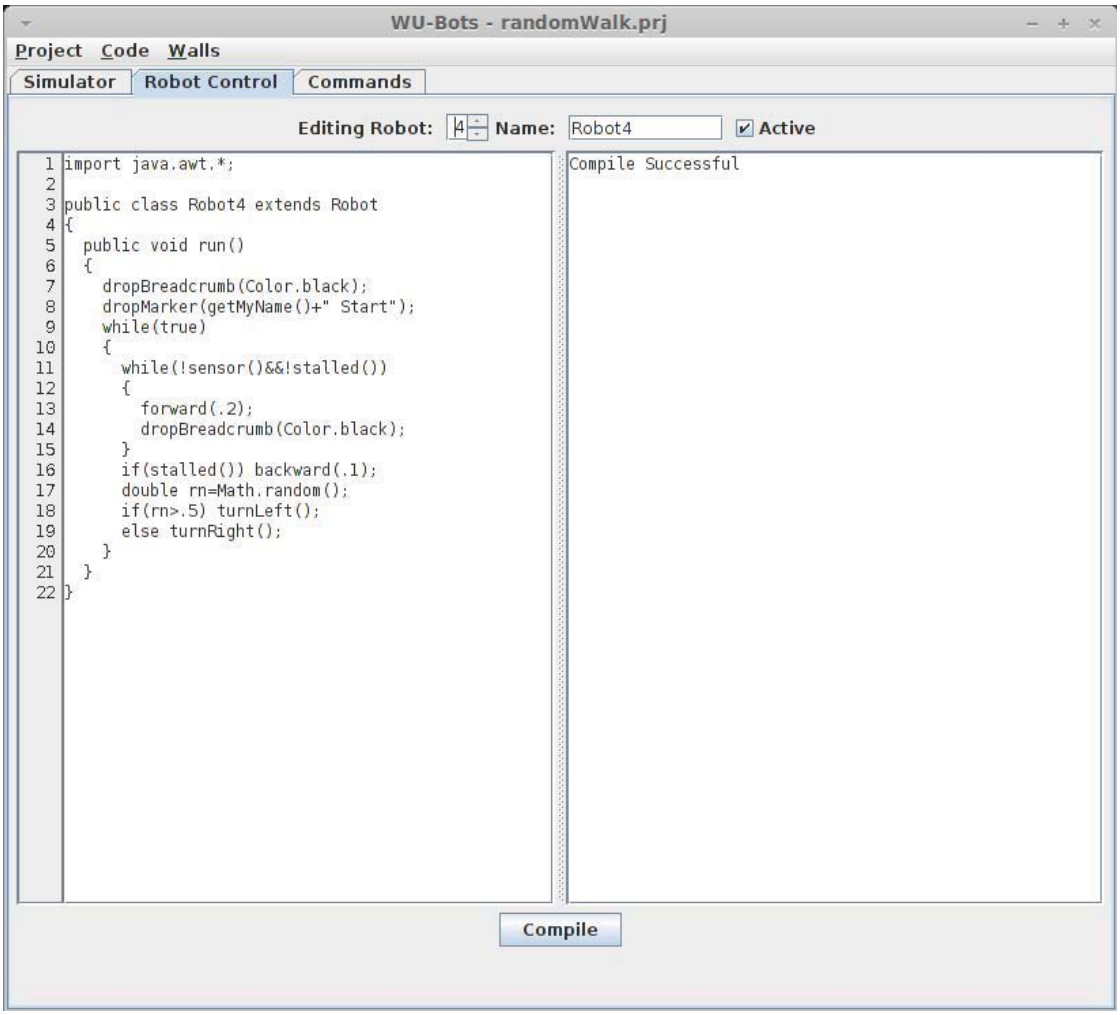

Figure 2. The Robot Control Tab

DESIGN

Many problems needed to be addressed when developing the simulator. A brief list is below.

1. How do we compile the student code from within the program?
2. How do we run the student code within the simulator?
3. How do we keep the robots synchronized in the simulator?
4. How do we detect collisions with the walls and other robots?

The student code is compiled using the "exec(...)" method of the Runtime class. One can use this method to execute any OS command from inside a Java program. We first write the student's code into a Java source code file (in the current directory where the simulator was launched). Each Robot is in a file called RobotN.java where N goes from 0 to 9. Then we issue the "javac" command through the exec method of the Runtime class. This returns a "Process" object which has an error stream that we use to capture any error messages to display in the user interface. Note that the Java development environment must be installed, and the path to the bin directory must be in the path environment variable.

To solve the second problem, the resulting class file is then loaded with a custom class loader. Each robotN class extends the Robot class which itself extends Thread. The method calls to control the robot are defined in the Robot superclass. After each class is loaded the "start()" method is called causing each robot object to execute as an independent thread.

The synchronization issue is solved by introducing a new thread whose sole purpose is to monitor the array of robot threads making sure that each one has completed its current step. When a robot object calls one of the robot methods (such as forward(1.0)) that command is broken down into a set of 20 millisecond steps which are then processed in a for loop. After each iteration of the loop a condition variable is set to indicate that this particular robot thread has updated its position. Each robot then calls the "wait( )" method. After all threads have updated the synchronization thread will check the system clock and when the next 20 ms interval arrives it updates the user interface, resets all condition variables and calls the "notifyAll( )" method. The robot threads thus run lock-step at exactly 20 millisecond intervals.

The fourth problem is solved by creating a candidate position for each robot and checking to see if that candidate position violates any collision criteria. If there is a violation the current position is not updated. Otherwise the candidate position becomes the new current position. For robot-robot collisions it is simply a matter of checking to see if the centers are at least two robot radii apart. For walls it is more complex. Vector analysis is used to find the intersection point from a forward-facing vector at each wheel and every wall. If an intersection is found, the distance to that point must be greater than a robot radius if the new position is to be allowed.

STUDENT PROJECTS

The simulator can be used for student projects at all levels, from the first introductory course to advanced courses on computational intelligence. In what follows we list ideas that we hope will challenge students at various skill levels.

Simple Project 1: Make the robot follow a square path in an environment with no walls. Have them then change the code to make the square larger, or follow a rectangular path. Have them drop breadcrumbs as the robot moves to show its path.

Simple Project 2: Make the robot follow a circular path, dropping breadcrumbs as it moves. Ask the student to make the circle larger or smaller. Have them change it to make a half circle. A figure 8.

Simple Project 3: Add walls to the environment. Make the robot walk a straight line until it comes up to a wall, then turn right (loop back here). They can use the "sensor( )" or the "stalled( )" methods or both. Then ask students to have the robot randomly turn right or left when it hits a wall.

Simple Project 4: Using a predefined maze, have the student write code so the robot (mouse) can move from a starting location to an ending location (cheese) in the maze. Perhaps have a contest to see which robot can move through the maze in the least time (the clever student will use diagonal paths when they can).

Intermediate Project 1: Make the robot trace a complex path in the environment such as tracing a triangle or spelling out a word with breadcrumbs.

Intermediate Project 2: Make the robot find a path through a simple maze by always turning the same direction. In human terms this is equivalent to always keeping the right or left hand on a wall. Show the student that this won't work with more complex mazes.

Intermediate Project 3: Starting with a random walk (Simple Project 3) have the robot calculate its position by tallying its forward or backward motion and angle. The robot should be able to reliably determine its position after any number of moves.

Intermediate Project 4: Have the robot store its moves in an ArrayList so that it can retrace its steps after some complex (perhaps random) sequence of steps.

Advanced Project 1: Have the robot walk through a maze and create a "node" every time it hits a wall. The nodes are then stored in an ArrayList that represents a graph of the maze. Make the robot drop a unique marker at each node. Over time the entire maze should be explored (except for regions unreachable by this method).

Advanced Project 2: Have the robot walk through a maze creating nodes at every change of direction. Have the robot keep track of what directions have been explored for each node, and return to all unexplored directions from each node at some later time. This can be done in a depth-first or breadth-first manner.

Advanced Project 3: Make the robot in Advanced Project 2 write a file with the graph of the maze in it. Make another project where a robot can load the graph.

Advanced Project 4: After the robot can load a graph of a maze, have the robot follow a shortest path algorithm to move from one node to any other node in the graph.

A project file illustrating some of the advanced projects can be found here [4]. The robot explores the environment creating nodes every time it turns. It also looks for co-linear nodes and reconnects the graph incorporating them. It then finds intersections of edges and adds nodes there. It prompts the user to enter a node and travels to that node using a shortest path algorithm. A screenshot of this project is shown in Figure 3.

CONCLUSION

The WU-Bots simulator was designed and implemented over several semesters with the help of several students. The result is an easy-to-use, executable Java jar file that will run on any computer that has the Java development environment installed. It provides an easy way to introduce the topic of robot programming at the introductory level, but also provides programming challenges to advanced students.

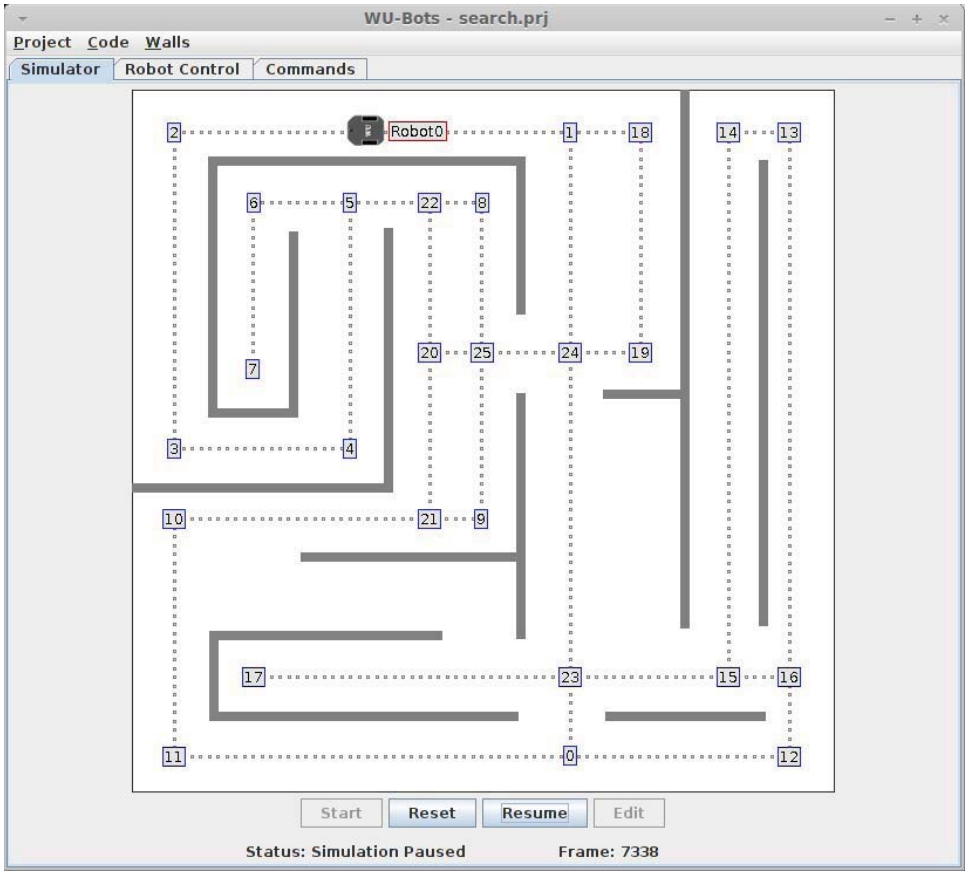

Figure 3. The Search Project

REFERENCES

[1] Follow this link for a summary of robot simulators available: http://en.wikipedia.org/wiki/Robotics_simulator

[2] http://en.wikipedia.org/wiki/Robotics_simulator#Open_source_simulators

[3] http://cislinux2.washburn.edu/wubots/WU-Bots.jar

[4] http://cislinux2.washburn.edu/wubots/search.zip. The Node and Path classes must be placed in the directory where the simulator is launched. They cannot be included in the robot code as private nested classes.