Session 2258

A Portable Mobile Robot Simulator for a World Wide Web Robotics Practicum

Steven J. Perretta, John C. Gallagher

Department of Computer Science and Engineering
Wright State University
{sperrett, jgallagh}@cs.wright.edu

Abstract

In recent years, courses in the design and programming of mobile autonomous robotics have been introduced at a number of institutions. These activities provide experience in a number of practical areas, including computer programming, project management, and technical writing. Further, they provide those experiences in an entertaining manner that may motivate students to pursue additional education in computer science and engineering. By their nature, however, these classes are resource intensive, often limited access to a few, fortunate students. In an experimental attempt to increase access to these opportunities, we have offered an introductory level course in autonomous robotics over the World Wide Web. In our class, students developed robot controllers to solve a series of increasingly difficult problems on a mobile robot simulator that we designed and implemented using Java. When finished, they upload their controllers to a real robot in our lab and observe the results via a WWW web cam. In this document, we will focus on our class' infrastructure with particular emphasis on the design and operation of a platform independent graphical simulation of the Khepera mobile robot. We will discuss how this freely available software provides accurate simulation, ease of use, and compatibility with the real robot in our lab. The paper will conclude with a discussion of the future plans and a set of open questions we intend to address in future offerings of the course.

1. Why WWW Autonomous Robotics?

Although formal classroom instruction is necessary to the education of engineers, it is not alone sufficient. Engineering is about solving problems of practical import. In reality, such problems are rarely as well defined as the average classroom exercise — which is usually designed to approximate problems one will really face. Real problems often lack a clear statement of what is needed. It can be as hard to figure out what to do as it can be to figure out how to do it. Real problems rarely possess any one "correct solution". Real problems, in short, defy domestication.

Recently, engineering practica employing autonomous robots as a pedagogical tool have become fashionable[1,2,3,4]. These experiences often employ the "low threshold no ceiling philosophy" which states that participation should have few prerequisites — but should allow for unlimited sophistication and complete flexibility in selecting solutions. In these courses, the idea of the practicum, defined by Schon as "a setting designed for the task of learning a practice"[5], is well realized. Students are afforded an opportunity to develop engineering skills that complement the analytical methods being learned in other courses. Further, designing and programming autonomous robots well satisfies the philosophy of "low threshold, no ceiling". Many autonomous robotics problems can be solved reasonably well using techniques accessible to the novice — yet are sophisticated enough to require advanced techniques to arrive at excellent solutions. Among other benefits, the above mentioned robotics based practica provide the following advantages to students:

a) The use of robots forces students to deal with problems that are rich in complexity. Wheels and gears slip, sensors fail or give spurious readings, and the environment in which the robot exists is usually somewhat unstructured — potentially introducing a huge number of unanticipated difficulties.

b) Students are actively engaged in deciding what they need to learn and how to learn it. Rather than being "taught at", they are encouraged to find answers and develop techniques and methods as needed.

c) Students are forced to learn how to communicate their ideas and validate their solutions. As in the real world, a major portion of the battle is to explain what one did and why others should believe it works. Since there is no one solution to any problem — there can be no answer key other than what the students themselves create.

d) The "low threshold" philosophy allows and encourages early development of practical design experience. This complements coincident attainment of specific domain knowledge and the development of analytical techniques.

e) The "no ceiling" philosophy allows limitless expansion, ensuring that the more able students are continually challenged and ensuring that no student will run out of problems to solve.

In our course, we provided students with a Khepera robot simulator upon which they could test ideas and verify code. We also provided remote access to an Internet connected mobile robot (see Figure 1) that was managed by another program we developed specifically for this class. Both pieces of software were written in Java, incorporate identical APIs, and employ very similar general interfaces. Students were expected to design their code using the simulator, and then test and tune their programs on the real robot. In this paper, we will focus largely on the technical and infrastructure challenges involved in developing portable simulation software that accurately modeled the Khepera robot, while maintaining an API that was both intuitive, easy to use, and allowed controllers to be seamlessly ported to software that interfaced the actual robot.

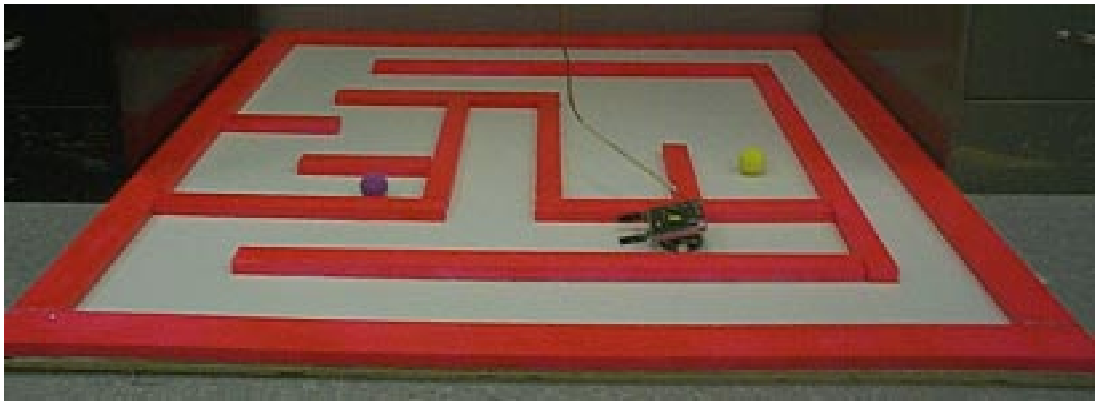


Figure1: An Internet Connected Khepera Robot

This snapshot was taken online with the WWW remote control camera. Users can use this camera to zoom in on any portion of the maze for detailed views. Another camera, not shown, provides a streaming video view from above.

2. The Robot Environment

The environment we created in our lab consists of a single robot that operates within a 4x4 foot enclosure. The robotic hardware consists of a standard Khepera robot[6,7] equipped with an auxiliary gripper arm module. Communication with the robot is facilitated through a wire tethered between the robot and a host machine's serial port. The robot is manipulated using software that writes/reads data to/from the robot via interpreted commands from the user. Within its enclosure, the robot may be confronted with a simple set of obstacles: reconfigurable walls (typically in the form of mazes and/or rooms), lights, and plastic soda pop bottle caps. Wall sections and lights are fastened to the floor of the enclosure, and therefore cannot be moved by the robot. Caps are used specifically as objects to be manipulated by the robot via its gripper attachment.

It is this particular environment that is simulated by our software. Due to the environment's simplicity, the task of developing sensor and actuator models was significantly reduced. The color and reflective properties of the obstacles were specifically chosen so that sensor response would be similar at given distances from an obstacle regardless of its type. These properties along with the constant lighting in our lab provided the basis for the accurate yet efficient models eventually used within the simulator.

3. Simulation Software Requirements

The primary constraints on the features of our software were defined by both pedagogical and practical concerns. An important practical issue we had to address was that of portability. Given that the simulator was going to be used by students working at home, the software had to be able to run on a variety of platforms including Macintosh, varieties of Unix, and MS Windows. Another consideration involved the complexity of the interface – both in terms of the GUI and the controller API. Our main goal in this regard was to create a programming environment that could be easily understood by relatively naïve undergraduates. The program had to be designed in such a way as to isolate the user from the internal details of the simulation and provide a simple, intuitive programming interface. Along these same lines, the graphical interface to the program had to be extremely simple; requiring only about an hour to learn. A third requirement dealt with integrating aspects of the simulator with our remote access software. This effort focused on providing a means to execute controllers developed under one environment in the other. The final and most important requirement we had to address regarded the development of internal models that simulate aspects of the Khepera robot. Aside from the obvious need to provide an accurate simulation of the robot, we had to consider simulating the typical interaction between the user (via a program) and the robot through a serial port connected line. The time intervals between user requests and the robot's subsequent response had to be relatively equal between the two programs. In order to adequately address these issues, we needed to define a reasonable compromise between the accuracy and the efficiency of the modeling implementations used.

4. Simulation Software Features

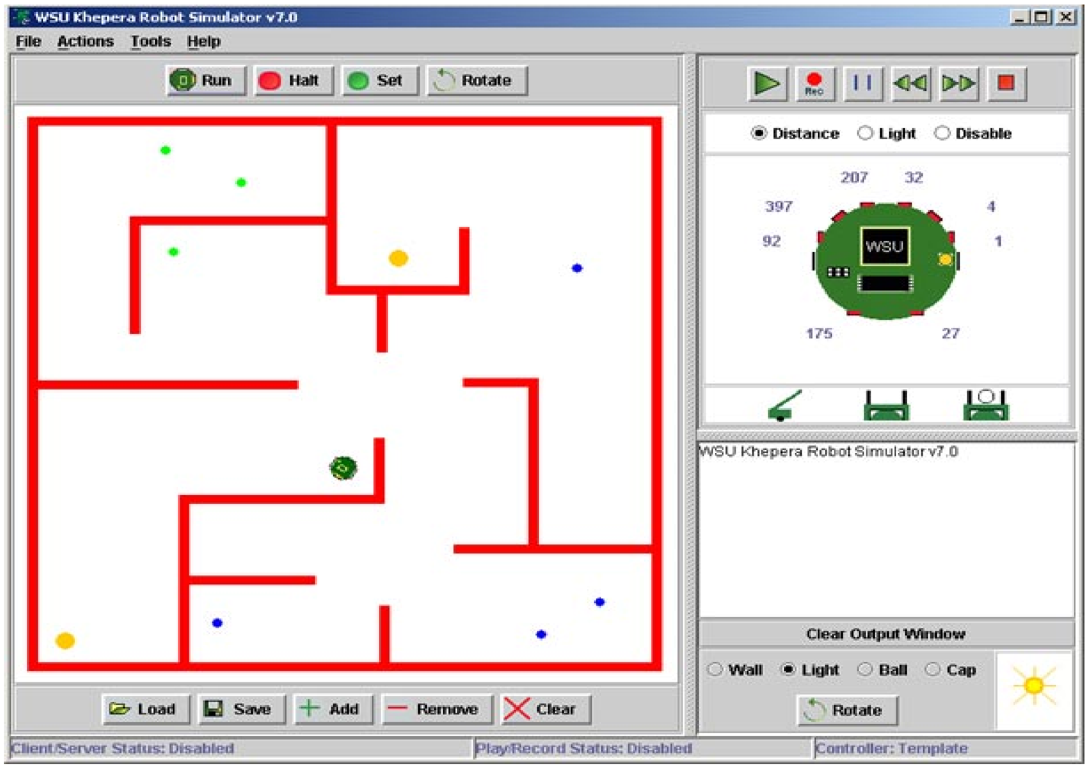
The current features supported by our software were incorporated to accommodate the specific properties of the robot's physical environment in our lab, and to satisfy other requirements mentioned above. What follows is a brief listing and description of the main features present in the current version of the simulator. Particular attention is paid to how the various practical and pedagogical constraints were addressed.

4.1 Basic Features and Functionality

As mentioned previously, the WSU Khepera simulator's front end consists of a graphical user interface that presents a variety of interactive control and visual references depicting the current state of the robot and its environment during the execution of a user defined controller (see Figure 2). Visual references are drawn in the main panel, where a 2 dimensional image of the robot and the 4x4 foot environment are maintained. Within this panel the user can interactively edit the environment by dragging and dropping objects. Another panel displays current sensor readings for both proximity and light sensing – providing real-time feedback to the user. Aside from other basic functionality (i.e. starting, pausing and terminating controller execution for example), the interface provides a means for users to modify certain simulation parameters. Both light and proximity sensor parameters can be manually adjusted to account for sensor degradation and fluctuating light sources. Program parameters used to simulate the robot's motors can also be adjusted to recalibrate the internal model in light of motor degradation or periodic malfunctions due to dust collecting in the gear systems. Other features include the ability to record controller runs for future replay, and the ability to execute a controller on multiple computers running the simulator via a secondary client/server interface.

4.2 Portability

The issue of portability was essentially two-fold: we required that the simulator be portable to a variety of PC related platforms, and that controllers be portable between our two pieces of software. Java was chosen as the implementation language for both the simulator and the remote serial port access programs, primarily because the Java virtual machine has been developed for most popular platforms. Besides being a highly portable language, Java offers other important



WSU Khepera Robot Simulator v7.0

Client/Server Status: Disabled
Play/Record Status: Disabled
Controller: Template

Figure 2: The WSU Khepera Simulator's Graphical User Interface

Here the robot is depicted in the main panel exploring a maze. In the panel on the on the upper right hand side, proximity sensor values for each of the robot's 8 IR sensors is displayed. At the bottom of this panel, a set of three icons depict the current state of the gripper – arm elements: arm, gripper, and gripper sensor.

benefits: reliable thread support, a full range of graphical components for GUI development, and an extensive API for creating 2 dimensional graphics. When employing Java, however, one also has to consider potential drawbacks associated with the language. For example, interfacing the robotic hardware is difficult to do without the use of separate packages that define APIs for serial and parallel port communication (i.e. Java Comm). In addition, Java is not well suited to system level programming, and we were forced to incorporate third party packages (in our remote access software) that enabled us to handle signals, manage serial port access, etc. Other potentially problematic features of the language are addressed in the following subsection.

Cross-program portability was achieved by incorporating a similar internal structure of class interactions within both programs. Specifically, this involved creating and maintaining a set of globally accessible objects that described the current state of the robot and pending commands to be sent to the robot's effectors. This set of shared objects forms the API, which is identical in both programs. A reference to these objects is maintained by both the controller and the main program, thus communication between these parts of the program is facilitated through indirect exchanges of information (the details of this method of interaction is deferred to the next section). In many ways the two programs are identical: the only real difference between them can be seen within the procedures used to derive sensor readings and set the robot's effectors.

4.3 Ease of Use

As the first step towards simplifying the interface, we wanted to create a controller programming environment that completely isolated the user from the inner workings of the program. To accomplish this goal, we defined a single controller class that executes as a separate thread. The controller is nothing more than a simple subclass of Java's basic Thread class (see Appendix). This thread is given access to the "virtual robot" using references to the objects that comprise the API. The execution of this thread is controlled by the main program through buttons displayed on the GUI, where the user can start, stop and pause the controller, and even dynamically load one of many controllers that may be available. Thus the user is spared from having to implement any form of thread management within the controller class itself.

To further simplify the programming interface, the number of classes forming the API, and the methods they define were kept small. This aspect of the program's design was actually easy to implement given that the Khepera's hardware components and command protocol are quite simple. There are three core classes defined by the API: the Sensor class, Gripper class, and Motor class. Each of these classes maintains pieces of data pertaining to the hardware component it represents. Classes that define an effector (i.e. Gripper and Motor classes) provide methods that allow the controller to manipulate components, such as setting motor speeds or raising the gripper's arm. Sensor related methods simply return the current state of the hardware component.

It should be noted that since the user's controller class essentially becomes part of the progam when it is executed, poorly implemented controllers can potentially effect the performance of the entire program. Most of the problems we have experienced, in terms of student developed code, deal with issues related to garbage collection. Periodic interruptions in execution due to frequent garbage collection by the JRE can be introduced by a controller that poorly manages locally created objects. Most, if not all, the problems we have encountered were in some way related to this issue. To remedy the situation, we recommend that students not familiar with Java understand the overhead incurred by creating and "loosing" objects.

4.4 Modeling and Accuracy

The development of the simulation engine required the design and implementation of internal models that articulated the Khepera's sensors and actuators in an accurate yet efficient manner. These models had to account not only for the average behavior of these devices as observed through controlled testing, but also for the periodic noise associated with sensors and motors. Calibrating values for both light and distance sensing between the physical and simulated robot was accomplished through extensive testing, which involved placing the robots in identical situations and fine tuning the simulation models. The resulting sensor and motor models effectively match the behavior observed with the Khepera, and because they are reasonably efficient, updates during simulation happen quickly.

A general model of the robot's interaction with its physical environment also had to be designed; this reflected situations such as the robot getting stuck against static objects, collisions with moveable objects, and gripper interactions. Static objects, such as walls and lights, present obstacles that will hinder the robot's progress if a collision is detected. If the robot's trajectory intersects one of these object types the robot will stop on contact, and it is up to the controller programmer to effectively re-orient the robot. When incidental contact is made with caps, on the other hand, the object will slide out of the robot's way – depending on the current trajectory. Gripper interactions are also modeled: caps can be grabbed, carried and dropped at random locations.

5. Implementation Notes

The core of the simulator was designed using a central thread to continuously loop through the key steps of the program. This thread serves as the backbone to the simulator engine (see Figure 3). Three main steps are executed in the engine thread: 1) the robot's position and orientation are updated based on current motor speeds, 2) sensor data is generated and stored in globally shared objects accessible to the controller, and 3) updates are made in the GUI that reflect the state of the robot based on changes in the first two steps. As these steps are executed, the controller thread uses all the derived data to plot its next move, and essentially runs synchronously with the simulation engine. The following subsections will provide a detailed look at what actually transpires at steps 1 and 2 of the algorithm. Step 3 will be ignored because its details are irrelevant to the main topic.

5.1 Position and Orientation

At step 1, the robot's position and orientation are recalculated. Data pertaining to speed is retrieved from the Motor object and tested against the speed values retrieved during the last iteration. If the speeds are the same, then the robot's position in Cartesian coordinates is updated along the same trajectory previously calculated. Because speeds are more likely to change over time periods much greater than the time taken for each iteration of the main loop, the series of calculations made on this data that are used to determine the robot's new coordinates can be short circuited when redundancies occur. The new Cartesian coordinates calculated are stored in a Robot object that is accessible to other internal objects that calculate sensor values and draw the robot on the GUI.

At times, the robot may be in a position in which it cannot move - regardless of its motor speeds. A collision flag is always returned by the call made in step 2 to evaluate sensors. This flag can take on a number of different values that specify if a collision has been detected and if so, what part of the robot has made contact. The collision flag is passed to the method called in step 1, where it is evaluated after determining the robot's trajectory. If the current path of the robot moves it in a direction facing the side where contact was made, then no update to the robot's position is made.

Wheel position updates are also generated at this step, and the resulting values are stored in the Motor object. A positive/negative motor speed increments/decrements the wheel position by a value proportional to the actual speed value. The position value is determined by both the speed value and the time acquired from a global clock, based on data collected from the Khepera. Time studies were taken determine the number of position increments/decrements at each second for different speeds. Because position values must be whole numbers, and due to the fact that the time taken for each iteration is much less than a second, position values are updated in chunks at certain global time cycles. For example, at the slowest speed the Khepera will increment a wheel position by 1 approximately every second. This means that at each call to step 1 during the engine's iterations, the amount to increment the wheel position will be some fraction of 1, and this value cannot be added to the position at this time. Therefore a

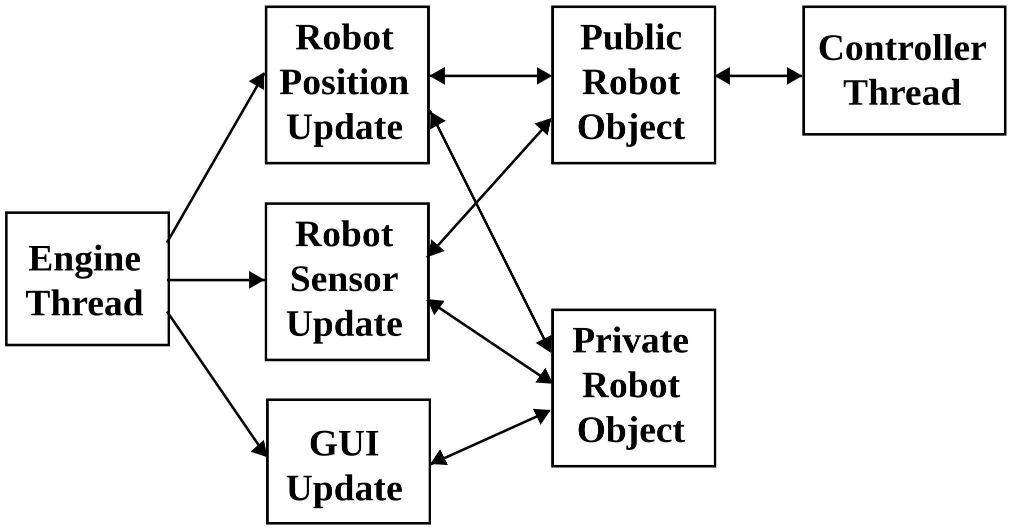


Figure 3: The engine thread's execution as it relates to data updates and controller interaction.

static timer is set, which is polled at each iteration of the engine. When a second passes (roughly speaking), the position is updated by a value represented by the current speed. Although the resolution of these updates, in terms of time, may not exactly match the real robot, the accuracy of the values themselves match quite well. In all the controllers developed by students in the class so far, the level of accuracy needed when using wheel positions was very minimal. In most cases, wheel positions are used to determine if the robot is stuck.

5.2 Sensor Updates

In step 2 of the simulation engine's iterative execution, sensor values are updated based on the new position of the robot. As noted earlier, light and proximity sensor values are updated automatically. Proximity sensors are always recalculated at this step. The main reason for this is given by the fact that almost every controller the user will develop, that defines some reactive behavior, will require continuous access to current proximity sensor readings. Even in the event that the robot has not moved, these sensor values are still updated. This is done because periodic noise may still creep in to sensor generated data at any time.

Proximity values are generated by superimposing the coordinates of "sensor beams" onto a matrix representation of the current environmental map. This matrix is actually a 500 x 500 array that contains values associated with the pixels that make up the environment display panel. Instead of actual pixel information, each value in the matrix corresponds to a number that uniquely identifies the type of object present at that particular coordinate in the map. Zeros are used to denote locations that are covered by the floor. Static objects like walls and lights use the same id number for each occurrence of their objects. For example, all points containing wall are denoted by a 1, and all points containing light are denoted by a 2. Caps each have a unique id. Because they can be pushed or carried by the robot, each cap has to be uniquely identifiable so it can be redrawn after being moved. Sensor beams that overlap values greater than zero in the matrix are taken to be in range of the object occupying that space.

Sensor values are then calculated based on the point of the closest intersection of the beam and object to the robot. If the object detected has an id number greater than 2 (i.e. it is a cap), and it is perceived to be within 5mm of the robot when the arm up, then it is added to a list of caps that may be subject to a collision. If the arm is down, caps in range of the front sensors are added to the list since they may be subject to a gripper grab.

When all eight proximity sensor values have been determined, a collision detection routine first determines cap collisions based on the robot's current state; that is, whether the arm is up or down. Using the coordinates for the robot's current position in the map and the location of the particular caps in the list, a test is initiated to determine if the areas containing the two objects overlap. This is done by using the Java 2d objects that define the cap and the robot in the display panel's drawing routines. The cap object, based on its id number, is obtained from the global set of graphics primitives that contain all the individual objects contained in the environment. Using a method from Java's 2DGraphics API, overlap between robot and cap can be determined. When a collision takes place, the cap is moved and the environment matrix is updated. The matrix update simply involves deleting the values associated with the cap, and assigning the cap's id number to new locations in the matrix. The new location occupied by the cap is determined by the trajectory of the robot and the location of the contact relative to the robot's outer surface.

When the gripper arm is down, then the cap(s) that were in range of the front sensors are tested to determine whether they are within range of the gripper's object sensor, or subject to collision with some part of the arm. The coordinates of the lowered arm are defined by a small set of points. An overlap test involving the cap(s) and these points is done in the same manner described above using built-in Java 2DGraphics methods. Caps that are involved in a collision are repositioned using a technique similar to one employed during collisions with the robot's body; depending on the trajectory of the robot and the point on the arm where contact was made, the cap is "shoved" in the appropriate direction. Caps that escape collision are subjected to a simple test that determines if they are within range of the gripper's object sensor. This sensor is implemented on the Khepera as a single IR sensor positioned on one side of the gripper facing the other side. The opposite side has a reflector, which directs the infrared beam back to the sensor on the other side. Thus, anything breaking the plane between the grippers is detected. In simulation a line between the grippers defines this sensor's range. The cap test simply involves finding any intersection between this line and the cap. If the cap intersects, then the sensor value in the global Gripper object is set, and a reference to the cap's graphics object is locally maintained. This reference is checked each time step 2 of the simulation engine is executed. If this reference is not null and the gripper is closed, the cap is considered to be in the possession of the robot, at which point it is removed from the environment matrix.

Light sensor values are actively generated only when light objects are present in the environment. When no lights exist, a constant value corresponding to relative darkness is multiplied by a random noise value that has only a 2% probability of being unequal to one. Noise levels unequal to one only deviate from this mean by a maximum of plus/minus .005. This formula reflects the relative stability of light sensor data in the absence of direct light. When lights are present, a calculation is first made to determine if the light is obstructed from the robot's view, or is out of sensor range. These tests serve as a short circuit evaluation to prevent needless computation. Range is first tested by calculating the absolute distance between the robot's centroid and that of the light(s). Any lights that are out of range are removed from consideration. Obstructed lights are determined by traversing a dotted line from the robot's centroid to any light in range. Points along the line are compared to points defined in the environment matrix. If a point on the line coincides with a nonzero value in the matrix, then the light is eliminated from consideration. Only every 7th point in the line is determined, to minimize the number of calculations made. Because the maximum width of any object in the environment is 8 pixels, the points along the line will always find an obstacle if present. The specifics of light sensing are deferred to the 'Light Sensor Model' subsection.

6. Simulator Usage

All of the software developed for our class is available for public download at our class web page [10]. By the time this paper appears, both software packages will be distributed with full open source (the licensing scheme is yet to be determined). After downloading and installing the simulator software, students can immediately begin to learn the interface by loading and running some of the demo controllers provided. Because these controllers come pre-compiled, the only extra software needed at this stage is some version of the Java run-time environment. The next step is to start writing one's own controller, which requires a Java SDK. Currently there are numerous implementations of the Java SDK; the most notable versions being distributed freely by Sun Microsystems and IBM. To make contoller development as easy as possible, a template controller source file is provided containing skeleton code and comments describing where/how to insert user code. A program manual and an online help system are provided as well, offering plenty of documentation to the user regarding the GUI, APIs and general controller development.

What follows is a list outlining the set of usage steps taken by students. The process of developing robot controllers is an iterative one, therefore these steps are usually repeated many times.

1) Write controller code. This is obviously easier said than done. In the early phases of development, this step usually involves writing a simple controller that does nothing but poll sensors as the robot moves in a circle or straight line. Students new to idiosyncracies of the Khepera robot usually need to first experiment with "dummy" controllers that give them a sense of what motor speed, light sensor, and proximity sensor values empirically relate to. As the development process continues, the code is refined to effectively deal with problems encountered in test runs. For most controllers, regardless of the specific goal, the later refinement stages deal with creating methods and defining helper classes that deal with sensor noise and recovery subroutines.

2) Compile controller code. Because every controller will contain references to objects and classes associated with the simulator's API, control code has to be compiled in the same directory as the simulator's class files. The compilation process requires javac.

3) Run controller code. In this step students start the simulator, then load and run their controller from the GUI. During the development process, a single controller may be run many times in order to reveal any bugs or logical errors in the code. The recommended approach to running tests is to start the robot at different locations in a given maze configuration, and/or run a given controller in multiple maze configurations. This strategy effectively places the robot in a variety

of situations in which it must successfully perform. Very often a controller may appear to be robust when the robot starts at a specific location and orientation within a particular environmental configuration. However, when the same controller is started under different circumstances, errors are often revealed.

4) Repeat steps 1-3 as necessary.

Example assignments for the class include implementing controllers to: visit all locations in a maze (maze solving), gather objects found initially in one “room” in the maze and move them to another “room” in the maze, and follow light beacons to traverse a maze. (See the Appendix for an example controller implementation).

7. Student Survey

One of the primary goals of our class was to introduce students to the potential pitfalls faced when interfacing with hardware, via autonomous controller design and implementation. The design of an effective and robust controller essentially involves two stages: a) develop an algorithm that can solve the problem at hand, and b) augment this algorithm with “safety” checks that cope with sensor/effector noise, and subroutines that either avoid or recover from potentially dangerous situations (e.g. getting stuck against a wall). Even the simplest control problem posed to the students required them to deal with the “imperfections” inherent in the hardware. For most students, this issue offered the greatest challenge. The most ingenius solutions to control problems could still suffer at run-time if the controller lacked the necessary mechanisms for dealing with the random periodic spikes in sensor readings, or the potential failure (or slow down) of a wheel motor. The upside to this ubiquitous problem, according to the students, was that once these issues were addressed in a generic fashion, the resulting methods could be reused in other controllers. In most cases students eventually developed methods for filtering out erroneous sensor data by taking a small series of sensor readings and measuring the distance between consecutive readings in the given sample, thereby determining which readings constitute abnormal values that deviate too far from the expected range. Designing a statistical model for normal sensor activity required considerable testing on the part of each student, and we were not surprised when we discovered that many students found this process to be the most tedious part of the class. However, once these issues were addressed, and students could focus on specific control problems, the feedback was very positive.

8. Future Considerations

Because the software was targeted at an audience consisting of undergraduates taking our specific class, it may not necessarily be useful as a general purpose autonomous robotics tool. There are three components missing in the simulator that, if added, would broaden its potential audience significantly: 1) a robot editor, 2) a new environment editor, and 3) multiple, concurrent robot support.

There are numerous robots available to researchers and the general public. Most of these differ substantially from the Khepera in terms of size, wheel configuration, and sensor type and position. Adding the capability to model other robots, or design one from scratch, would obviously broaden the simulator’s applicability.

The environment and the objects that compose it are directly correlated to the robot arena in our lab, as of the current version of the simulator. The scale of the enclosure and the object types that can be introduced to it are hard-coded into the program. Adding an environmental editor to allow users to scale this enclosure to alternative dimensions and provide the means to custom design objects and object collision physics, would obviously make the simulator much more attractive to those operating robots in different environments.

Multi-robot collaboration has become a hot topic in robotics and artificial intelligence research. Incorporating multiple robot support into the simulator would have the greatest impact on the program's usability compared to the additions mentioned above. Unlike the more drastic re-engineering of the code required by the above modifications, multiple robot support could be easily integrated into the existing program given the recent addition of client/server support.

9. Summary

Many of the features incorporated into the software were derived from the specific needs of the autonomous robotics class. These features include: 1) cross-platform portability of the simulator, 2) cross-program controller portability, 3) a simple, intuitive API that reflects the basic functionality of the Khepera robot, 4) a simple controller programming environment that insulates the user from the underlying implementation, 5) gripper-arm support, and 6) a realistic model of robot/environment interaction within the simulator.

Cross-platform portability was addressed by using Java as the implementation language, as mentioned above. Robust thread support and light weight GUI components serve to add extra benefits to using this language.

Seamless porting of controllers between both programs was achieved by developing a broad based design common to both the simulator and remote access programs. Although the underlying implementation of each program is significantly different, each provides the same API and controller thread class to the user.

The API was designed to provide the user with a simple and complete set of classes (and associated methods) for interacting with the robot. There is no difference between requesting and acquiring sensor data, for example. From the user's perspective the difference between communicating with the simulation engine and communicating with the robot is absent. Because the controller is implemented as a thread, it can be considered a "separate program" by the user. This approach serves to divorce users from the rest of the program's inner workings, and helps the programmer to better visualize the execution of the controller. These features help to give focus to what needs to be done to solve a problem rather than how to implement it within the context of the program.

The WSU Khepera Simulator effectively models the interaction between the robot and its environment. This is done efficiently using generated proximity sensor values to short circuit collision detection. Since sensor values are computed all the time, this data can be used to determine when collision detection routines should be executed - as opposed to constantly calling these routines. This leads to an overall efficiency gain, and allows the use of more accurate and realistic collision detection algorithms because they are used only periodically.

For more information on our autonomous robotics class and Khepera targeted software see[8,9].

Acknowledgements

This work was supported by the National Science Foundation under Grant No. 0096311. Additional support provided by Wright State University and the Ohio Board of Regents. The authors would like to thank Richard Drushel and Wendy Peluso for their help in our research efforts.

Bibliography

1. Beer, R.D., Chiel, H.J., and Drushel, R.F. "Using Autonomous Robotics to Teach Science and Engineering", Communications of the ACM (June 1999). ACM Press.

2. CWRU Autonomous Robotics Course. Online. http://www.eecs.cwru.edu/courses/lego375/

3. Martin, F.M. A Toolkit for Learning: Technology of the MIT LEGO Robot Design Competition.

4. MIT 6.270 Autonomous Robot Design Competition. Online. http://www.mit.edu:8001/activities/6.270/home.html

5. Schon, D. Educating the Reflective Practicioner : Toward a New Design for Teaching and Learning the Professions (1987).

6. K-Team (Khepera Info). Online. http://www.k-team.com/

7. Mondada, F., Franzi , E. and Ienne, P. "Mobile Robot Miniaturization: a Tool for Investigation in Control Algorithms", ISER'93, Kyoto, Japan, October (1993).

8. Gallagher, J.C. and Perretta, S. "WWW Autonomous Robotics: Enabling Wide Area Access to a Computer Engineering Practicum", The Proceedings of the 33rd Technical Symposium on Computer Science Education. ACM Press (2002).

9. Perretta, S. and Gallagher, J.C. "A General Purpose Java Mobile Robot Simulator for Artificial Intelligence Research and Education", Proceedings of the 13th Midwest Artificial Intelligence and Cognitive Science Conference (2002).

10. WSU Autonomous Mobile Robotics course. Online. http://gozer.cs.wright.edu/classes/ceg499/ceg499.html

Appendix

The following source code serves as an example of a typical student's effort to implement a controller that enables the robot to completely navigate through a maze. Given that the maze was constructed with continuous wall segements (i.e. no "island" maze sections), the strategy used by this student involved following and maintaining contact with the wall to the left side of the robot.

```
/*
 * Robot.java
 *
 * Created on July 27, 2001, 11:06 AM
 */

/**
 *
 * @author  bill
 * @version
 * ———————— Wall Follower ————————
 */
public class Robot extends Thread {
     // Do NOT Modify these fields
    private CurrentRobotState currentData;
    private MessagePasser messagePasser;
    private Sensor[] sensors;
    private Motor motorState;
    private int gripperState;
    private int armState;
    private int resistivity;
    private boolean objPresent;
    private boolean active;

    // Add your own fields here....
    private int distThreshold = 600;


    /** Creates new Robot */
    public Robot(CurrentRobotState data, MessagePasser mp) {
        currentData = data;
        messagePasser = mp;
        active = true;
    }

    private void getRobotObjects() {
        sensors  = currentData.getSensorValues();
        motorState = currentData.getMotorState();
    }

    private void setForwardSpeed() {
        motorState.setMotorSpeeds(3, 3);
    }

    private void setSpeed(int left, int right) {
```

```
    motorState.setMotorSpeeds(left, right);
}

private void testProximity() {
    int max = 0;
    int closeIndex = -1;
    int closeIndex2 = -1;

    for (int i = 0; i < 5; i++) {
        int tempVal = sensors[i].getDistValue();
        if(tempVal > distThreshold) {
            if(tempVal > max) {
                max = tempVal;
                closeIndex2 = closeIndex;
                closeIndex = i;
            }
        }
    }
    if(closeIndex == 0) {
        if(closeIndex2 != 2 && closeIndex2 != 3)
            motorState.setMotorSpeeds(3, -1);
        else
            motorState.setMotorSpeeds(3, -5);
        return;
    }
    if(closeIndex == 1) {
        if(closeIndex2 != 2 && closeIndex2 != 3)
            motorState.setMotorSpeeds(3, 1);
        else
            motorState.setMotorSpeeds(3, -5);
        return;
    }
    if(closeIndex == 2 || closeIndex == 3) {
        motorState.setMotorSpeeds(3, -5);
        return;
    }
    if(closeIndex == 4 || closeIndex == 5) {
        if(closeIndex2 == 2 ||  closeIndex2 == 3)
            motorState.setMotorSpeeds(3, -5);
        else
            motorState.setMotorSpeeds(-1, 3);
        return;
    }
    motorState.setMotorSpeeds(1, 3);
}

private void findWall() {
    setSpeed(3,3);
    while(sensors[2].getDistValue() < distThreshold)
        sleepTime(100);

    setSpeed(2,-2);
    while(sensors[0].getDistValue() < distThreshold)
        sleepTime(100);
        setSpeed(0,0);
    }


    private void sleepTime(int time) {
      try {
          Thread.sleep((long)time);
      } catch (InterruptedException e) {}
    }

    public void killControl() {
        active = false;
    }

    public void run() {
        getRobotObjects();
        findWall();
        while(active) {
           testProximity();
        }
    }
}
```

STEVEN J. PERRETTA
Steven Perretta is currently an instructor/Ph.D. student at Wright State University's College of Engineering. His research interests include autonomous robotics, pattern recognition, and artificial olfactory sensing.

JOHN C. GALLAGHER
John Gallagher is an assistant proffesor of computer science and engineering at Wright State University. His research interests include neuromorphic computation, evolutionary algorithms, autonomous robotics, and internet enabled engineering practica.