Simulating a LEGO Mindstorms RCX Robot in the Robotran Environment

R. Mark Meyer and David C. Puehn

Canisius College
Computer Science Department, 2001 Main Street WTC 207, Buffalo, NY 14208
meyer@canisius.edu, dpuehn@gmail.com

Abstract

LEGO Mindstorms robots are very popular with colleges and universities for teaching computer concepts and programming. These robots elicit excitement in students and provide a nontrivial, real-world platform for exploring algorithmic concepts. We created a simple algorithmic language, called Robolang, and wrote a translator that turns it into Lejos code, a variant of Java that can be run on the RCX version of the LEGO Mindstorms robots. Seeing that students were eager to explore programming with the RCX robots at home, we wrote a graphical simulator to visualize actions of our penbot, a configuration of the RCX robot that we used in most assignments. Using an emulator approach, we intercept the ROM calls to the RCX's hardware made by the TinyVM, the stripped-down Java Virtual Machine that runs compiled Java bytecodes. Our system then forwards these calls to a software model that represents the actual robot hardware. The software model creates the graphics to mimic the penbot using Java2D. This approach greatly simplified coding by capitalizing on existing software, namely the Java compiler and the JVM. Students can program either in Robolang or in actual Lejos and use the simulator to visualize the actions of the robot acting as a sort of visual debugger.

Introduction

In 2005, eager to get students to program the RCX LEGO Mindstorms robots in a simplified procedural language, we created a simple environment called Robotran (Meyer and Burhans 2007). A programming language called Robolang was embedded in Robotran, providing the ordinary algorithmic constructs as well as robot movement commands. Robotran contains a parser that creates a Lejos program from the Robolang program, which can then be compiled using lejosc and downloaded to a real robot. We shied away from the primitive visual programming language that LEGO provided, called Robolab, because of its weaknesses and because we wanted the students to begin experiencing a programming language that was not too far removed from Java which they would see in the following semester.

Robotran was very successful at our college. Students enjoyed it and were able to draw complex letters and shapes, follow the beam of a flashlight, sense a black track on paper underneath the robot, and respond to touch sensor bumps using only a few lines of code. Lejos, which is Java with a special robot-command library, was deemed too complicated for this group of students, some of whom were taking robotics for a general education requirement. However, Robolang was sufficiently powerful that students could program for our robot competition in it and do quite well against other students who wrote directly in Lejos or NQC (Not Quite C).

A limiting factor with all robots used in education is the expense of the robots. Since each kit costs approximately $300, we couldn't expect students to buy their own, especially if they took the course for general credit. To avoid loss or damage, we could not lend out robots, forcing students to only work on robot assignments at school, but not at home. It was then that we decided to embark on the ambitious route of simulator building.

Since the purpose of our simulator was very specific and curricular in its scope, we didn't attempt to create a physically accurate general-purpose simulator, but rather one that would respond to our students' Robolang programs somewhat realistically. Our penbot is a standard roverbot (Fig. 1) with bump sensors, a pen attached to a motor for up and down movement and a light sensor. To match our curricular use, we modeled our simulator's robot on this hardware configuration, adding a world of barriers, light sensors and a floor on which the robot could draw and a track that the robot could follow.

Because we needed to run compiled Lejos programs, which are actually Java programs, we wrote the simulator and IDE in Java. This permitted a conservation of effort since we emulate the josx API with our own Java code, instead of writing a simulator engine that follows the Robolang program directly. This strategy appears to be unique to our approach and has a number of benefits as well as trade-offs which we will explore.

Writing and testing a simulator is a very daunting task. Though more testing needs to be done, the simulator is

mature enough to let students do most of their assignments with it.

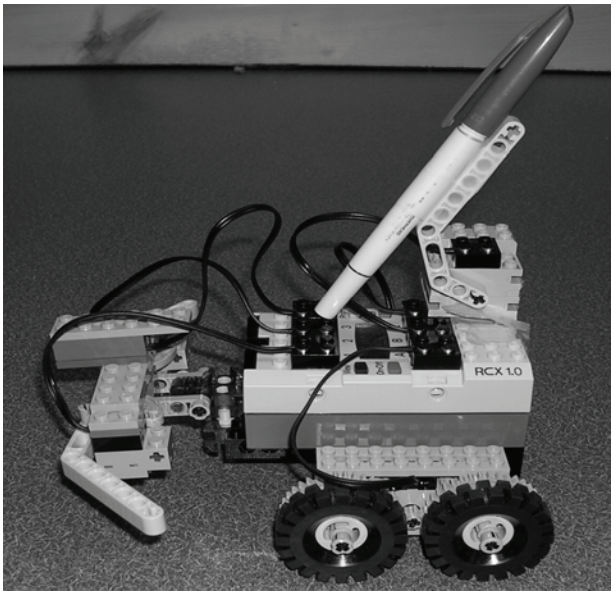

Fig. 1: Penbot used in Robotran

Previous Work

Robot simulators fulfill a crucial role in saving time and money, both in industrial and educational settings. Simulators make it possible for students to explore high-cost robots that they would likely never get to touch. They enable more students to write and debug robot controller programs, unconstrained by the limited number of robots. There are many reasons for using simulators as well as some drawbacks (Dodds 2006).

There exist several dimensions along which a robot simulator can be pegged to describe crucial differences between simulators. These include 1) tie-in to a real-world robot, 2) dimensionality of the graphics, i.e. 2D or 3D, 3) physical verisimilitude, 4) configurability, 5) programming model and permitted programming languages, and 6) number of robots in the world and 7) obstacles in the world other than robots.

Karel the robot is one of the most widely used robotic simulators, although it does not actually simulate an existing robot, but rather an idealized one (Pattis 1995). The programming language Karel, and its descendent, Karel++, was based on structured programming languages e.g. Pascal, and later C++ and Java. Since the purpose of Karel is to teach programming, verisimilitude is less important. Karel is part of a long lineage of robot-like teaching tools, going back to the LOGO programming language and others. LOGO was designed to implement turtle graphics, which itself was an idealization of turtle robots.

With the advent of low-cost robot kits about ten years ago and their continued improvement, the field has seen an explosion of interest in programming actual devices rather than simulators. Simulators are still useful as test beds or to provide access to otherwise expensive machinery. Along with this shift in emphasis to real hardware, simulators are now driven by the hardware. Thus, the tie-in to physical robots has strengthened since the days of LOGO and Karel as more projects attempt to mimic the real world.

An important dimension of simulator differences is whether they attempt to model the physical properties and constraints of a robot accurately. The Doane Roverbot simulator attempted to do that (Buss et al. 2005). Its creators studied the roverbot's physical movement, including its speed at different power levels. They carefully measured the motion of the actual roverbot using a motion sensor and charted its position versus time at ten different power levels. The authors also measured the torque exerted by the motors and factored in the weight of the robot to make the most realistic 3D simulation possible. However, they found that some movements, like turning, were very difficult to model. They fell short of implementing a collision detection system or sensor inputs.

However, some point out that simulators are often too perfect (Dodds et al. 2006). They do not replicate real-world settings in which robots do not move in a perfectly straight line or respond to every sensor input in a consistent way. The Jago simulator (Wolf et al. 2003) corrected this by causing the simulated robots to sometimes veer off course and to represent quantification errors that arise from sensors, thereby giving students a more realistic taste of what they can expect from actual robots.

One of the most visible dimensions of difference among simulators is the level of the graphics. Robotran is only 2D, as is Jago and Karel. Gazebo offers both 2D and 3D simulations, and runs on Linux only. The Doane simulator is a 3D simulation. Many commercial packages, such as Webots, use 3D graphics for a more realistic simulation. Commercial simulators offer a much larger range of functionality but are often not free. Vajta and Juhasz (Vajta and Juhasz 2005) discuss the role of 3D simulation in robotic design, test and control. They note the difficulty of achieving true binocular vision and the need for special glasses or at least visual clues that give the illusion of depth. Their Webots Simulator is widely used and they discuss a newer simulator called RobotMAX that uses a 3D visualization engine and a CAD interface to allow the user to design virtual environments (Tellez and Angulo, 2006).

Yet another dimension of simulator design is configurability. Robotran permits only one modification of

the basic roverbot design, specifically the orientation of the light sensor. Commonly, other educational simulators, such as Jago and Doane, do not permit much customization. Some commercial packages, including Webots, permit the user to define a model, which is a configuration of sensors and motors. This model is uploaded to the simulator for use during actual runs and is not alterable during execution.

The programming model is where Robotran differs from all other LEGO simulators. Most simulators accept programs written in some programming system that is used with actual robots and then interpret the commands in terms of the simulated world and robot. Assembly language is sometimes used, as in RCXSimulator (Butler and Brockman 2001) NQC (Not Quite C) is another popular choice.

Some simulators, such as the Doane simulator, emulate the brick by interpreting the low-level bytecodes. This approach gives more control over the simulation but also means that the software must include a full interpreter for the bytecodes, either P-script or Java. Writing and testing this part of the simulator, which is behind the scenes, is a huge undertaking in itself. Robotran makes use of the standard Java bytecode interpreter, the JVM, which is available for almost every computer, and intercepts the specific commands to the RCX brick. This approach was not seen to be duplicated elsewhere, at least in terms of RCX simulators.

Some RCX simulators, for example LegoSim (Roblitz, Buhn and Mueller 2002), model the behavior of the brick alone. They show an image of the brick, as LegoSim does, with clickable button images and the LCD display, but there are no sensors or motors visibly attached. LegoSim displays three boxes for the three sensor input ports, and three other boxes for the A, B and C motors, displaying the direction and speed. The emphasis is on the internal state of the brick.

Other simulators model a brick with sensors and motors, forming a roverbot, penbot or other student-built robot, like the Doane Roverbot Simulator (Buss et al. 2005). This robot uses a set configuration which has two bump sensors and two motors. The world that the simulated roverbot moves around in is a rectangular box with some objects in it, closely mimicking a real-world test setup. The Doane Roverbot uses downloaded P-Brick script files, which are generated by the visual programming language Robolab, into the simulator which then interprets them and runs the simulation.

While it is not necessary to have more than one robot in a simulated world, having the capability of more than one allows exploration of inter-robot behavior, for example "sumo wrestling" or racing, popular themes for robotics competitions. Jago permits multiple robots per world, whereas Robotran does not.

User Interfaces

Robotran contains two major windows: an IDE for editing Robolang and Lejos code, and a simulator for visualizing what the code will do on a real robot. The IDE permits entry, editing and saving of programs in either language. It also translates from Robolang to Lejos. Thus, the simulator can be used in our earliest course where only Robolang is used, as well as our CS 1 course, which is a Java-based introduction to programming. Fig. 2 shows the basic IDE with tabbed panes in which the programs are edited.

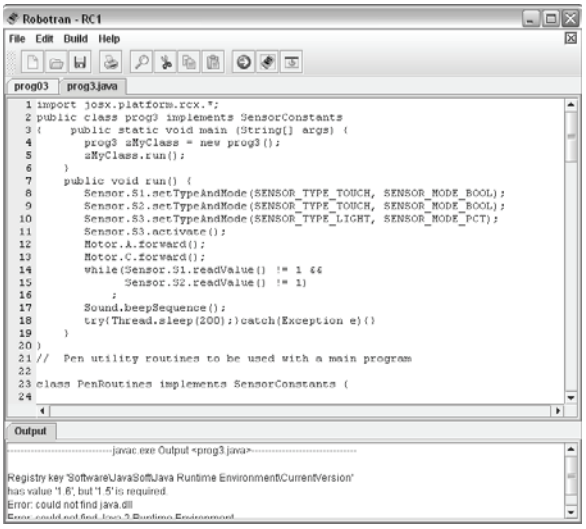

Fig. 2: Robotran IDE

The simulator's visual interface consists of two panes: the virtual robot in its world, and an image of the RCX brick with control buttons (Fig. 3). The buttons on the image can be pressed just as a student would press the buttons on a real RCX to start, run and stop a program. The LCD also provides output identical to a real RCX robot.

The robot's world is simple, consisting of a square sheet of paper over which the robot moves and draws if the pen is down. Only one robot can exist in this world. There are four boundaries around this world which the robot can hit and by which it will be stopped. Additional barriers can be placed inside the world. Blue barriers represent marks on the paper that trigger the robot's light sensor if the light sensor is oriented down. This allows students to test out line-following algorithms. A simulated light source can be placed in the world, corresponding to a flashlight which the robot may follow or avoid, depending on the assignment.

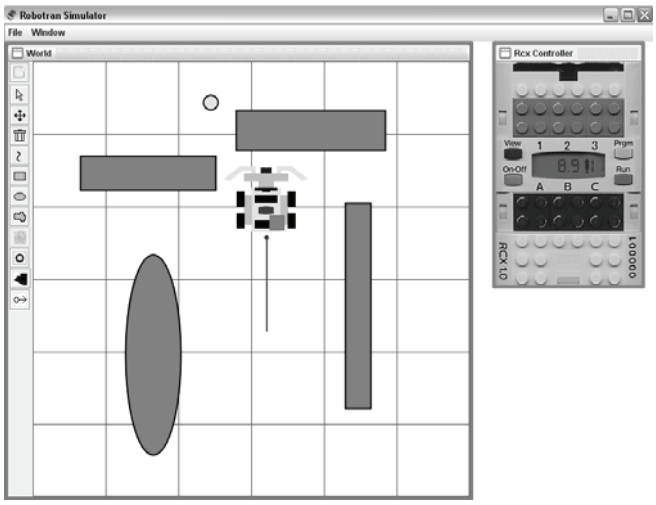

Fig .3: Robotran simulator interface

Simulation Architecture

The key to making this approach work is to emulate calls to the RCX's ROM. Normally, the TinyVM calls routines in the RCX's controller directly via Java JNI (Java Native Interface). JNI is the Java technology that permits calling machine language routines outside the JVM. The physical robot responds to these calls by turning motors on or off or actuating a sensor.

Our approach replaces the bottom layer only, namely the real robot, with a software model, comprised of a number of Java classes. This model's interface is the set of ROM calls that TinyVM makes. By replacing the original package that Lejos used on the robot with our own, we cause the running program to call our model instead of a real robot. Another thread continuously inspects the state of the model and makes calls on a graphics panel to display a visual analogue of what the real robot would be doing. For instance, if the TinyVM would have turned on motors A and C in the forward direction, our model's components corresponding to A and C motors would be set. During the next repaint cycle, our graphics object would get updated to show the simulated roverbot moving forward in a straight line.

The following steps comprise the lifecycle of a student RCX program as executed on a real robot:

1. Student writes Robolang code on PC.
2. Student translates code to Lejos using the Robotran translator on PC.
3. Student compiles the Lejos code on PC using lejosc.
4. Student downloads the .class file to the RCX using the infrared tower.
5. The .class file now resides in RAM of the RCX and the TinyVM is already in RAM, ready to execute.
6. Student presses the green RUN button on the RCX.
7. TinyVM loads the .class file and interprets the bytecodes, making calls to the hardware ROM when methods inside the josx package are called.
8. The student stops the program by pressing the STOP button on the robot.

The lifecycle of a simulated program differs from the above as follows. The .class file created from compiling the student's program exists only on the PC and is never downloaded to the robot.

1 to 3. Identical to above
5. The student starts the simulator software which loads the .class file into PC's RAM.
6. A graphical window opens, which continually checks the state of the in-RAM RCX model.
7. The PC's JVM interprets the bytecodes of the .class file, making calls to our stand-in josx package. The running program changes values in the RCX model. When these are eventually spotted by the graphics window thread, they cause the simulator to change its display.
8. The student stops the program by pressing the on-screen STOP button.

Fig. 4 illustrates the two complementary scenarios.

Most of the josx framework's functionality is achieved by performing native ROM calls on the RCX. This means that for a given Java operation the endpoint usually is a ROM call to the RCX. RCX ROM calls require a memory address and zero to four parameters. Robotran takes advantage of this simple hardware interface since all that must be done is to emulate the functionality of the RCX ROM calls.

Here is a sample Lejos program that starts the RCX's motors going forward indefinitely.

```
Motor.A.forward();
Motor.C.forward();
```

The Java code that these statements activate in our simulator is the following:

```
ROM.call(0x1a4e, 0x2000, 1, 3);
ROM.call(0x1a4e, 0x2002, 1, 3);
```

On the real RCX these calls would pass through JNI and the layers of native code and reach the hardware. Our simulator redirects these calls to our virtual ROM instead of the native code.

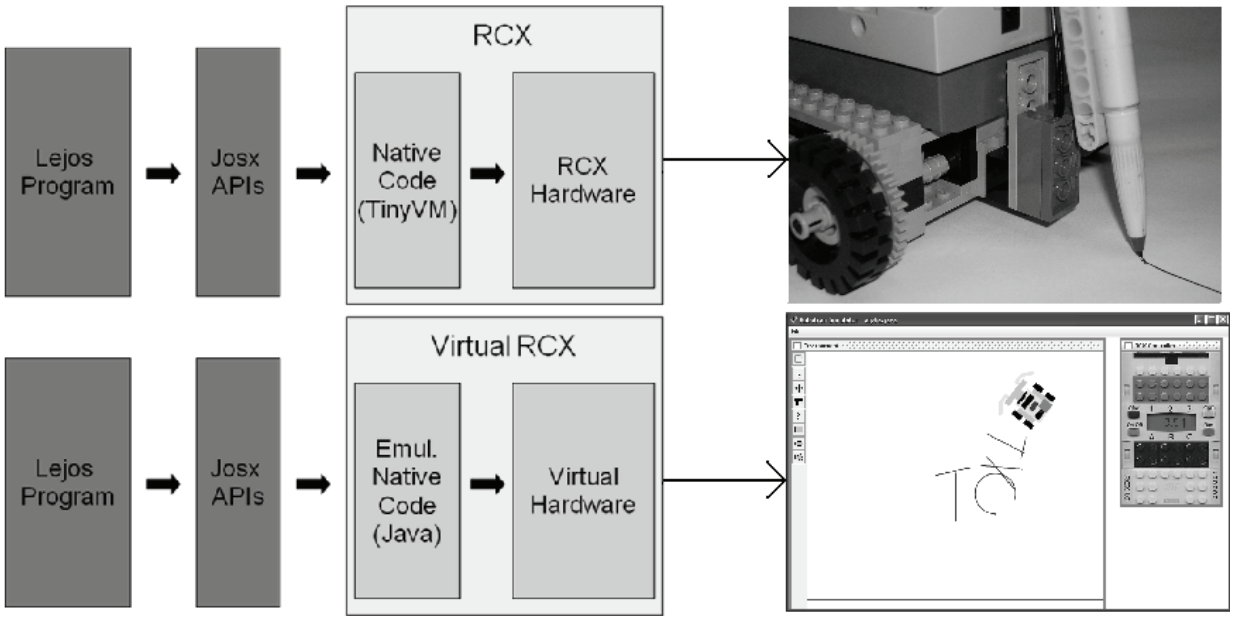

Fig. 4: Comparison of the actual lejos/RCX architecture with Robotran simulator

Here is the ROM method header from the original josx code:

```
public static native void call(short aAddr,
  short a1, short a2, short a3, short a4);
```

Next is our modified version of the method that redirects the call to our virtual ROM.

```
public static void call(short aAddr, short a1,
            short a2, short a3, short a4) {
   RCX.rom.call(aAddr, a1, a2, a3, a4);
   //RCX.rom points to Robotran's virtual ROM
}
```

Only minimal changes needed to be made to the josx source to intercept the ROM calls since all ROM calls are passed to a native function. Once the ROM call enters the virtual ROM it is handled by the following code:

```
//controlMotor call:
//a1: which motor
//a2: the motor's mode
//a3: the motor's power
else if (aAddr == 0x1a4e) {
       if (a1 == Opcodes.MOTOR_0) {
              rcx.motorA.setMode(a2);
              rcx.motorA.setPower(a3);
       }
       else if (a1 == Opcodes.MOTOR_1) {
              rcx.motorB.setMode(a2);
              rcx.motorB.setPower(a3);
       }
       else if (a1 == Opcodes.MOTOR_2) {
              rcx.motorC.setMode(a2);
              rcx.motorC.setPower(a3);
       }
}
```

The parameter of the call specifies changes to the state of the model, in this case the `rcx` object that replaces the real robot, by updating Robotran's internal data structures The example above changes the mode (direction) and the power of the virtual RCX's motors. There are data structures for every component in a real RCX, including Button, Display, Memory, Motor, and Sensor.

Alternative Approaches

Our need to simulate students' Robolang programs directed many of our choices. The first approach was to write an interpreter that would interpret and simulate Robolang directly. However, the interpreter would have required quite a lot of code. The biggest flaw of this design is that simulating at such a high level means we would not be able to simulate Lejos programs at all.

The current design of intercepting the RCX's ROM calls was not the first idea we tried. Other designs were considered and some were prototyped. One way was to create an intermediate language (IL) that the simulator would interpret. A benefit of this design is that we could simulate any program that could be compiled down to this IL. We spent some time with this idea and even designed the intermediate language. But it was abandoned because it would have required us to write an interpreter for it as well as a Lejos to IL compiler, a not insignificant task even using a compiler compiler like JavaCC or YACC.

Another design that was considered involved writing a Java bytecode interpreter that would interpret a compiled

Lejos program. This would give us much more control of the execution of the program since we would be able to precisely control timings and forbid features that exist on desktop Java but are not implemented on the TinyVM. We decided against this approach because the learning curve of understanding how to interpret Java bytecodes was too steep for this project.

Although our current scheme has some downsides we felt that it was the best choice since it required much less new code and was at a low enough level to simulate most Lejos programs.

Challenges and Future Work

Robotran is now stable enough to be used in a classroom setting and will be deployed in Spring 2009 with a CS 1 class. Consequently student feedback does not yet exist but a survey will be taken and grade comparisons with the last two years' worth of CS 1 students will be made to ascertain any visible effect of using the simulator. In those previous classes, the RCX robots were used for about half of the lab projects, but no simulator was involved.

Despite the elegant simplicity of our approach and the economies which we exploited, writing a simulator complete enough for students to use in place of robots is a substantial undertaking.

Several user interface enhancements are expected in the near future, including an ability to save a world configuration, which consists of the set of shapes, barriers, floor marks and light source, and then to reload it. A longer range goal is permitting more than one robot to exist in the simulation so students could hold competitions or investigate emergent behavior from multiple robots.

The minimal requirements for running Robotran are very modest. The Java 1.6 JDK must be installed because it calls the Java compiler as well as the JVM. We have tested Robotran on a number of average Windows PCs as well as a MacBookPro and a PC running Ubuntu Linux and performance was more than adequate.

The most pressing, and perhaps most difficult, challenge is to recast the simulator so it can work with Lejos NXJ and to model an NXT LEGO Mindstorms robot. The code for NXJ has been significantly restructured. Furthermore, several sensors exist in the NXT that were either rarely used with NXTs or not built in. Every NXT kit comes with a sonar sensor, which would have to be modeled in Robotran. This would not actually be too difficult because the simulator could duplicate the logic of a bump sensor, with the change in that a barrier or boundary does not need to actually touch the bump sensor in order to activate it. Even more significant are rotation sensors that are built into the NXT's motors. Though rotation sensors existed for the RCX, they were externally attached to wheels and used up valuable sensor ports. The NXT sensors are accurate to 1 degree of rotation, in contrast to 6 degrees in the older system. Because the NXT's rotation sensors are always present in the motors, they are more often used in NXT projects and should be simulated.

Acknowledgements

Kevin Mastropaolo, a junior at Canisius College, took over the project in September 2008 and finished several significant features, including collision detection, light sources and shapes on the floor for track following. He is currently working on porting Robotran to NXJ.

References

Buss, C., Gilbert, A., Paisley, N. and Sillasen, J. 2005. The Doane Roverbot Simulator. *MICS 2005: Proceedings of the Midwest Instruction and Computing Symposium*, Eau Claire, WI, April 2005.

Butler, J., and Brockman, J., 2001. A Web-based Learning Tool that Simulates a Simple Computer Architecture, *SIGCSE Bulletin*, Vol. 33, No. 2, 47-50, ACM Press.

Dodds, Z., Greenwald, L, Howard, A., Tejada, S., and Weinberg, J. 2006. Components, curriculum, and community: robots and robotics in undergraduate AI education, *AI Magazine*, Vol. 27, No. 1, 11-22, AAAI Press.

Meyer, R.M. and Burhans, D.T., 2007. Robotran: A Programming Environment for Novices Using LEGO Mindstorms Robots, *Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference,* 321-326*,* AAAI Press.

Pattis, R.E., 1995. *Karel The Robot: A Gentle Introduction to the Art of Programming, 2nd Ed.,* New York: Wiley.

Roblitz, T., Buhn, O., and Mueller, F., 2002. LegoSim: Simulation of Embedded Kernels over Pthreads, ACM Journal on Educational Resources in Computing, Vol. 2, No. 1, 117-130, ACM Press.

Tellez, R. and Angulo, C., 2007. Webots Simulator 5.1.7. Cyberbotics Ltd., *Artificial Life*, Vo. 13, No. 3, 313-318, MIT Press.

Vajta, L. and Juhasz, T., 2005. The Role of 3D Simulation in the Advanced Robotic Design, Test and Control. *International Journal of Simulation Modelling*, Vol. 4, No. 3, 101-156, DAAAM International.

Wolfe, D., Gossett, K., Hanlon, P.D., and Carver, C.A., 2003. Active Learning Using Mechantronics in a Freshman Information Technology Course, *ASEE/IEEE Frontiers in Education Conference*, 24-28, IEEE Press.