The Doane Roverbot Simulator

Cole Buss, Allen Gilbert, Nate Paisley, and Jason Sillasen
IST Department
Doane College
Crete, NE 68333
[cole.buss, allen.gilbert, nate.paisley, jason.sillasen]@doane.edu

Abstract

As part of Doane College's 2004 Summer Research Program, we undertook the task of creating a 3D simulator to run programs written for a Lego® Mindstorms™ "Roverbot." We coded our simulator in Java and Java3D because of their extensive documentation, cross-platform nature, and unbeatable cost. Using both freeware CAD programs and the Java3D API, we created models of the Roverbot and its test environment. We gathered physical data about real Roverbots and how they function in the real world, and then attempted to translate our findings into code that would appropriately affect our simulation.

Although our current simulator is not complete, our work has provided a good framework for future improvements, and the experience of creating such a simulator has increased both our knowledge and respect for software design.

1 Introduction

Over the past few decades, the importance of computer-aided simulation has been firmly established. From advances in space travel to improvements in city traffic control, the ability to simulate real-world events with virtual visualization has proven to be a time and cost efficient way to solve problems. The possibility to reap these analytical benefits of simulation provided excellent motivation for the creation of our Lego® Mindstorms™ (1) Roverbot simulator during our 2004 Summer Research Project. Our vision for the simulator was for it to allow development of Roverbot programs without need for the actual Roverbot hardware. We strove to provide a virtual environment for testing Roverbot control programs without having to repeatedly beam revisions to the physical robot, thus saving time and battery power. Furthermore, we wanted the simulator to allow multiple people to work on the same project and be able to individually test control programs without needing multiple Lego kits.

To better understand our project, one must first be familiarized with the concept behind Lego Mindstorms Robot Kits. In the words of Lego's website,

> LEGO® MINDSTORMS™ lets you design and program real robots that do what you want them to. With the Robotics Invention System 2.0™, the core set of the LEGO MINDSTORMS product range, you can create everything from a light-sensitive intruder alarm to a robotic rover that can follow a trail, move around obstacles, and even duck into dark corners. (1)

The operation of robots built with the Mindstorms kits is controlled by a microcomputer housed inside an "RCX brick." This yellow brick includes three motor control outputs, three sensor inputs, operational buttons, a digital display, and an infrared receiver. A user can create control programs using bundled Lego software that provides a visual representation of structured programming. Thus, the Lego Mindstorms kits are useful for teaching the basics of programming to beginners.

The Roverbot (pictured in Figure 1) is a simple robot with two different kinds of sensor inputs. The front of the robot houses two bumper sensors that can detect and prompt a reaction to a collision. Although not pictured below, our Roverbot also houses a front-mounted light sensor that can react to differences of dark and light in its path. Two motors drive both the front and back wheel on each side, providing mobility. Thus, turning is accomplished by giving more power to one pair of wheels over the other, or by driving the motors in opposite directions.

Our Roverbot's test environment is a 4 x 8 foot wooden "arena" with one-inch ridges on each side. Figure 2 pictures this environment with obstacles (for use with the bumper sensor) and blue tape (for use with the light sensor).

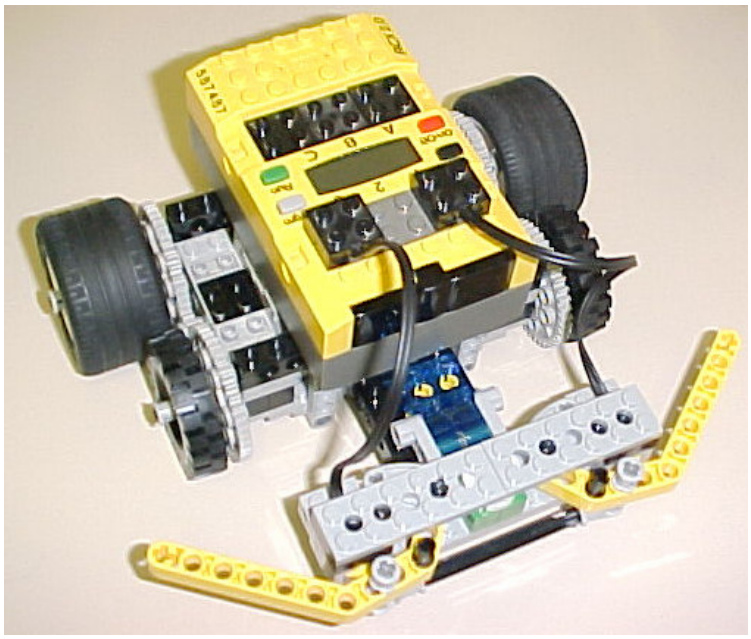

Figure 1: A Real Roverbot

Figure 2: The Roverbot Test Environment

In this paper we will discuss the tools we used to create our simulator, the methods we used to create our 3D content, how we modeled the behavior of a real Roverbot, our P-Brick Script code compiler and interpreter, the Graphical User Interface of our simulator, and our future plans for the simulator.

2 Tools

Our project leader and professor, Mark Meysenburg, decided that we would use Java 1.4 and Java 3D 1.3.1 (2) as our programming languages for the simulator because of their extensive documentation, cross-platform nature, and cost-free availability. Furthermore, we were introduced to NetBeans, a freeware Java development environment that provided, among other powerful features, a revision control system. This system allowed us to keep track of our latest versions of code, while multiple project members worked concurrently on the same project. We also used other freeware programs for content creation. These tools are described in the following sections.

3 Content Creation

After familiarizing ourselves with the basics of Java and the NetBeans development environment, we set out to create the "external" part of our simulator: 3D models of the Roverbot and its test environment. To construct our virtual environment, we simply mapped out the x, y, and z coordinates of each vertex of the physical test environment and then used that information to construct a 3D representation. We used the same process to make our rectangular obstacles and then added contrast to our models using the simple Java3D coloring system. As for lighting and shading, Java3D provided a default lighting scheme that suited our needs for a basic simulator.

3.1 CAD-Created Geometry

The creation of our Roverbot model was more involved, as we wanted to create a realistic-looking virtual representation. Thankfully, we were able to utilize a fantastic freeware program called MLCad (Mike's Lego CAD) (3). This program allowed us to create digital Lego models using a vast library of individual Lego pieces. These pieces were polygonal data files defined by another freeware program, Ldraw (4). Ldraw's website provided downloadable parts libraries as well as detailed information about the specifications of the Ldraw file format. By using these programs to create our model, we spared ourselves from the difficulty of having to define each vertex of each surface (as with the creation of our virtual environment). After completing our detailed Roverbot model, we saved it in the Ldraw file format and then converted it to a Wavefront .obj file using the LdrDat2 freeware program (5). We were then able to load the Wavefront file into Java3D objects.

Although the converter did a very good job of translating the vertices for use with Java3D, it was unable to convert the color data from MLCad into a form that Java3D could recognize. Consequently, we had to color the model by opening the Java3D model file in a text editor and replacing each reference to a color with a useable Java3D counterpart.

Another difficulty we encountered dealt with scaling. In the conversion process, our CAD-generated models were "blown up" so that they were much, much too large. Therefore, in our Java classes representing the Roverbot model, we had to scale the geometry down to make the Roverbot model match the dimensions of the physical robot.

Figure 3 shows the Roverbot model we created using this process. The model has a high level of detail, and a correspondingly high number of polygons to render.

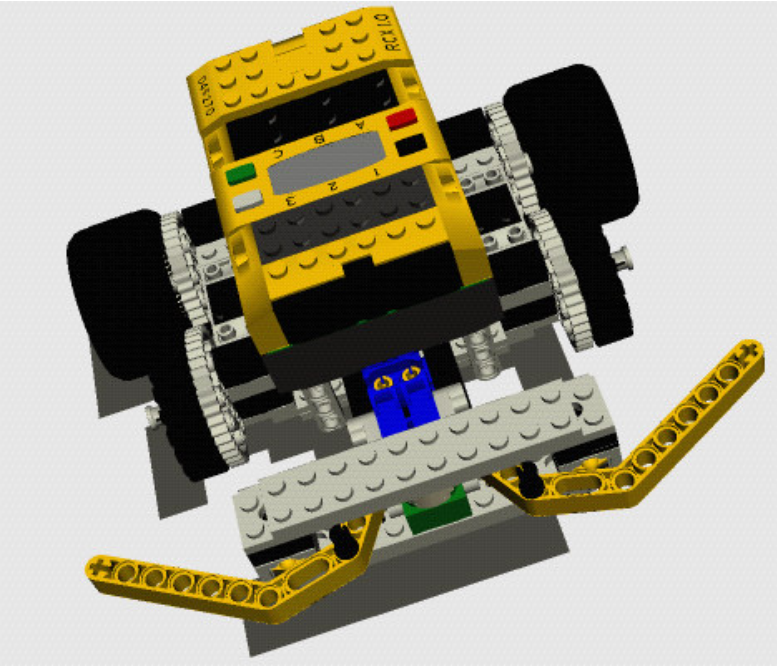

Figure 3: High-Polygon Roverbot Model

3.2 Hand-Coded Geometry

As mentioned above, some of our 3D models were "hand coded" rather than being developed through CAD software. Some of the models were simple enough to make CAD software overkill. In particular, our models of the 4 x 8 Roverbot "world" and the obstacles placed on the world were simple rectangular shapes. For these models, we made measurements and hand-coded vertex information directly into the corresponding Java classes.

Recognizing that lower-end hardware would not be able to effectively render our high polygon Roverbot model in a simulation, we also decided to create a low-polygon alternative. Initially, we created a low-polygon model by including only the CAD-generated RCX brick and bumpers, and by representing the wheels with simple cylinders. However, this still resulted in a relatively high polygon count, due to the numerous Lego studs on the RCX brick, the holes in the bumpers, and so on. To further reduce the number of polygons, we created another model using only hand-coded vertex

information. The resulting model, shown in Figure 4, has reasonable fidelity and loads much faster than the high-polygon model.

Another object we hand-coded was the shadow that appears under the high-polygon Roverbot model.

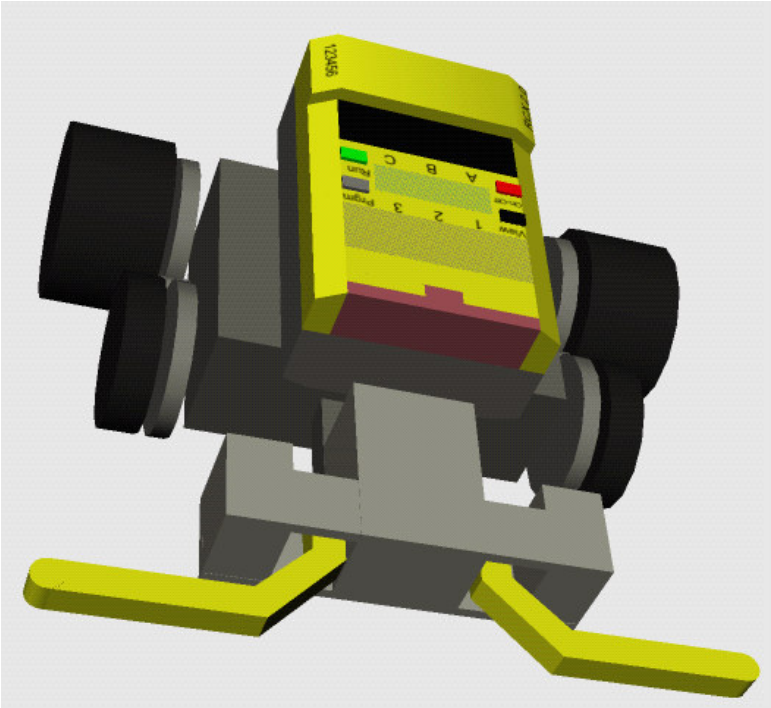

Figure 4: Low-Polygon Roverbot Model

4 Roverbot Behavior Modeling

After creating the "externals" for our simulator, we began the creation of a physics model. In order to accurately portray the Roverbot's movement, we had to study a real Roverbot in motion.

First, we needed an understanding of the speed of the robot at different power levels. Each motor could be programmed to rotate in either direction, at one of ten different power levels. To measure this aspect of Roverbot movement, we used a motion sensor provided by our physics department. We had the robot move towards the motion sensor at each power level, and from a position versus time graph displayed on a computer, we were able to calculate the speed of the robot at each level.

Next, we measured the torque exerted by the Roverbot's tires at each power level by connecting a force sensor to a single tire and using a computer to give us a readout of the measurements. We also calculated the total weight of the robot so that we would know the normal force exerted by the world on the robot. Using these measurements, we could

create a physics model to move the Roverbot accurate distances in the correct amount of time.

Rotating our simulated Roverbot was another challenge to tackle with our physics model. Rotation was tricky because the physical Roverbot would turn a different amount depending on the power setting of the motors. For simplicity, we only had one side of tires rotating (only one motor on) when calculating the angle of rotation. We placed the robot on a table and marked where each tire was. Consequently, we discovered that the robot was rotating around its rear stationary tire. Using a protractor, we marked the location of the tires after a one-second turn and then determined what angle the Roverbot had rotated through. We were then able to calculate how far the robot would rotate at each power level, given the time, using various trigonometric equations.

As mentioned earlier, we performed the turns with only one set of tires rotating. However, it is possible to make one set of tires rotate backward while the other set rotates forward, allowing for a sharper turn or a spin. Because an object rotates about its center of mass, we needed to calculate the center of mass of the robot. Using the measurements we took of the size of the RCX block and the tires, we calculated the center of mass of the entire object. We found that it was slightly to the rear of the robot (because of the large rear tires), but centered from side to side, because of the robot's symmetry. With this information, we were able to program our simulated robot to rotate about its center of mass when performing a "spin" turn.

5 P-Brick Script Compiler and Interpreter

Another key aspect of our simulator was the ability to load and execute programs developed in the Lego Mindstorms programming environment. Although programs are developed graphically in this environment, they are actually saved as plain text files. We developed Java classes to read these P-Brick Script code files, compile them into a custom byte-code format, and then interpret the byte-code to move the simulated Roverbot accordingly. In order to do this, we needed to understand the P-Brick Script language. Once we understood the language we could build the compiler and the interpreter.

We found that the P-Brick Script language has syntax somewhat similar to C. A simple program is shown in Figure 5.

We developed a pre-processor class to make compilation of the P-Brick Script programs simpler. The pre-processor removes comments and "#include" lines, expands macros, and adds whitespace so that the code is easier to parse.

After pre-processing, our compiler class converts the program into a custom byte code format. The byte code has commands for motor control, sensors, and timers, as well as mathematical operations, control structures, conditions, and so on.

A third class represents the Roverbot controller. This class interprets the byte code produced by our compiler and moves the simulated Roverbot accordingly.

```
program test {

     #include <RCX2.h>
     #include <RCX2MLT.h>
     #include <RCX2Sounds.h>
     #include <RCX2Def.h>

     main {
          ext InterfaceType "kRoverBot"
          rcx_ClearTimers
          bbs_GlobalReset([A B C])
          try {
               bb_Forward(A, C, 100)
               bb_TurnLeft(A, C, 100)
               bb_TurnRight(A, C, 100)
               bb_Backward(A, C, 100)
          } retry on fail
     }

}
```

Figure 5: Sample RCX-Code Program

Our P-Brick Script code compiler currently has basic functionality, but it is not complete. The sounds that can be generated from a RCX brick are not supported, for example. In addition, we were not able to incorporate sensors into the controller class.

6 Graphical User Interface

We created a simple Graphical User Interface for the simulator. The GUI allows users to load and execute programs, and also supports different views of the simulated world. The view can be changed to pre-set positions via menu options, or manipulated manually with the mouse and keyboard. A screen shot of the GUI is shown in Figure 6.

We wanted a splash screen that would allow the user to choose whether to use a low or high polygon Roverbot. We created an image for this splash screen by rendering our high-polygon model using a freeware program called POV-Ray (6). Our GUI and splash screen were created using the NetBeans IDE. The splash screen is shown in Figure 7.

Our GUI has basic functionality, but more features need to be added. We would like to be able to place obstacles and blue tape on the world, and to be able to pick up and move objects in the simulator. In addition, the manual view manipulation is currently

cumbersome. We would like to improve this in future versions of the simulator. These concerns are addressed in the Future Work section below.

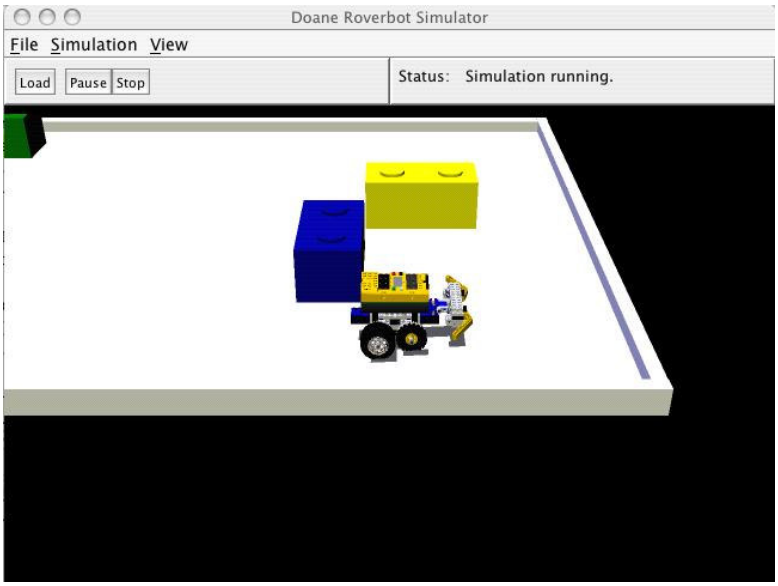

Figure 6: Doane Roverbot Simulator GUI

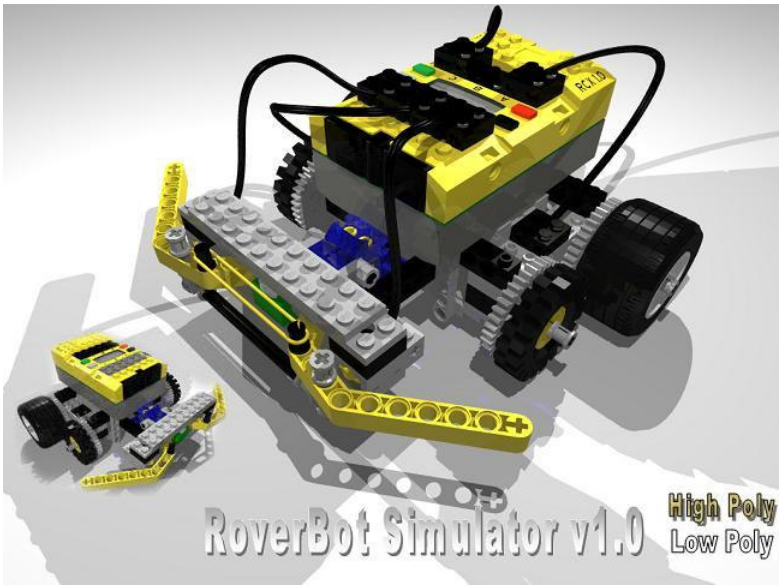

Figure 7: DRS Splash Screen

7 Future Work

There are several aspects of our simulator that we are planning to improve in order to more faithfully mimic the behavior of the physical Roverbot. In particular, we intend to implement a collision detection system in order to support bumper sensor events. We will also modify our interpreter to provide support for other sensor events (namely, the

light sensor) as well as other less common RCX functions. Additionally, we plan to improve the fidelity of the physics model underlying the movement of the simulated Roverbot. Finally, we want to enhance our Graphical User Interface with features that allow placement of obstacle blocks, placement of “blue tape” (for use with the light sensor), and more flexibility in terms of viewing angles presented to the user.

Work on the simulator will continue during the summer of 2005, in another Doane Undergraduate Research project.

8 Conclusion

This project provided us with many opportunities to learn through experience. We were able to learn much about programming in Java and Java3D using the NetBeans development environment. We were all introduced to a very important concept of group programming called "common code ownership," and used the NetBeans CVS (Concurrent Versions System) to keep our files up to date on a server. When using CVS, we learned how to resolve conflicts in different versions of code, how to submit updates, and how to manage our local development environment.

As for our graphical work, we were able to learn some new things about how computers process 3D models. We learned about 3D model construction, using both CAD software and hand-coding methods. We learned how to manipulate the look of a model by using a simple text editor to modify the code that represents the model's vertices and colors. When working with our models in Java3D, we learned how to orient objects in a virtual world and create animation behaviors.

We learned how to model real-world behaviors in a computer simulation, and we also learned about compiling and interpreting computer languages.

Over all, the experience of creating our simulator was very valuable. Participating in the software design process has increased both our knowledge and respect for software design, 3D modeling, compiler construction, and computer simulation.

References

(1) Lego Mindstorms Website: http://mindstorms.lego.com/eng/products/ris/index.asp. Link active as of 3/10/2005.

(2) Sun (Java) Website: http://java.sun.com/. Link active as of 3/10/2005.

(3) MLCad Website: http://www.lm-software.com/mlcad/. Link active as of 3/10/2005.

(4) Ldraw Website: http://www.ldraw.org. Link active as of 3/10/2005.

(5) LdrDat2 Website: http://wave.prohosting.com/xxcoder/ldrdat2.htm. Link active as of 3/10/2005.

(6) POV-Ray Website: http://www.povray.org. Link active as of 3/10/2005.