A New Simulator for Botball Robots
Stephen Carlson
Montgomery Blair High School (Lockheed Martin Exploring Post 10-0162)

A New Simulator for Botball Robots

1 Introduction

Simulation is important when designing and testing any engineering design, particularly a robot. Even though a simulation cannot exactly match reality, general concepts can be examined quickly and cheaply in a simulation. In particular, trying code in a simulator is extremely useful, since mistakes can be quickly differentiated from actual logical issues which need to be debugged on a real board. When accounting for the time taken to download code, charge robots, and reset the board, two or three simulated tests can be conducted in the time required for one real-world test.
However, it is difficult to find simulators well suited to testing Botball competition robots. KISS-C has a built-in simulation program that works on simple programs, but it cannot simulate non-Create robots or sensors. Its physics are also sketchy in terms of collisions with the PVC pipe, a technique often used to align a robot. Other simulators such as USARSim [1] have fantastic physics engines but are not well suited to the fully autonomous challenges of Botball. A new solution is needed for effective simulation.

2 A New Simulator

A useful Botball simulator should be able to:

1. Run unmodified robot code, without requiring even an extra kissSim_init().
2. Replicate real-world physics.
3. Adapt to changing board layouts.
4. Follow real robot behavior to reasonable precision and accuracy. While fine tuning would be required in any case, less is better.
5. Allow tests to be examined in detail postmortem, for the times when the developer's video camera pointed at the board is not good enough to find out what went wrong.

With these goals in mind, a new Botball-specific simulator was designed and coded from scratch in Java. Java's cross platform nature opens its use to a wide variety of systems and avoids locking out dedicated Macintosh or Linux fans. This project, started in 2008, had to be heavily modified twice to account for the changing controllers used in Botball, so it was not actually ready for use until this year. The simulator's main screen is shown below in Figure 1.

Figure 1: Screen shot of the simulator in action.

2.1 Features

Properly parsing C code is difficult, and re-implementing an algorithm to do so takes a long time to design and test. A solution comes in the form of the Java compiler. Since Java and C are similar, the Java compiler (which is available to Java programs) can be used to compile and execute a user program, thus gaining native Java control over the actions of the program. By "re-parsing" the Botball C code into Java code, a program is generated that can be run by the simulator, with calls to Botball library functions going through the simulator itself. This method works well enough for almost all Botball programs; the shortcomings section below goes into more detail on its failures.

The simulator accepts user C code when the "Load Code" button is clicked; pressing the green play button will start the code with the output from print statements shown in a text box on the bottom right. CBC buttons are simulated with either the keyboard or buttons on-screen. The table graphics shown on the screen are read from a file which can be specified on simulator start up, allowing the simulator to be reused for future years. Both basic CBC and iRobot Create robot designs can be simulated, with functionality similar to that of the built-in KISS simulator. Create wheel encoders and touch sensors work as expected, but the battery, cliff, drop, infrared, and wall sensors all return default values. The CBC robot has no sensors by default, but sensors can be easily added later.

Sensors are also simulated to a degree in the code. While KISS-C style button punching and slider dragging is allowed to manually simulate analog and digital inputs, additional methods have been added to test sensor-heavy code. For fans of normally-closed touch sensor designs, right-clicking a digital port at the bottom of the screen will flip the behavior of the port from normally open to normally closed and back. Ports can be associated with simulated sensors placed on the robot that can measure distance (ET), contact (touch), line color (top hat), or the starting light. Most Botball sensors have at least partial support, including the accelerometer, but the accuracy of the values is likely to vary from sensor to sensor and CBC to CBC. While support for sensors is still basic and a work in progress, having some sensors working is an improvement over having none. No more excuses for not using any sensors in the program!

2.2 Realism

The physics in this simulator are also more realistic than the built-in KISS simulator. Instead of stopping on impact with a wall (or worse, driving through it), the new simulator uses the forces provided by each motor to generate a separate force and a torque around the robot's center, causing the robot to exhibit more realistic behaviors such as sliding alongside a wall or rotating to line up with it. An example of such an improvement can be seen in Figure 2, where the left hand side shows how the KISS simulator might handle this scenario whereas the right side shows the correct behavior displayed by the new simulator. Such behaviors are often exploited by robots seeking to establish alignment with the unmoving field boundaries, as driving a robot with a flat back bumper straight backwards into a pipe will tend to align its bumper with the pipe, even if it starts slightly off angle. Fixing this behavior also accurately simulates other issues, such as the robot not instantly changing speeds or directions but instead having to change speeds continuously.

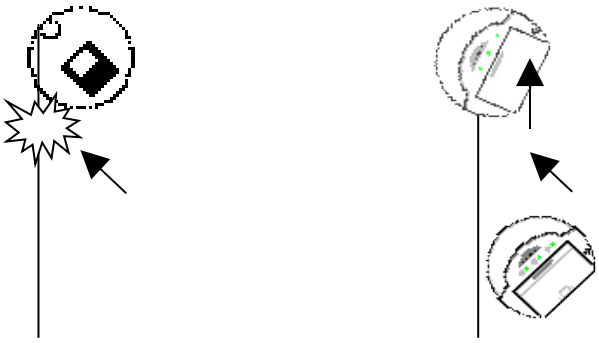

Figure 2: Sketchy physics versus realistic physics.

For the times when the robot does not succeed in its mission, the simulator provides tools to help find out what went wrong. Since this version of the simulator supports complete determinism, running the program again without clicking any of the randomize buttons will produce results identical to the failed run, allowing one to continually test the worst-case scenario while refining the program. A pause/play button allows the simulator to be paused or resumed at any time to examine precisely how much things need to be changed. For the

times when one is willing to violate the unwritten code of robotics for the sake of testing, the "Hand of God" button will allow divine intervention in test runs to reposition the robot.

2.3 Shortcomings

The biggest shortcoming of this simulator is its limitation to two robot models – a generic iRobot Create chassis and a generic CBC powered chassis. In reality, robots could be designed with other types of drive systems such as Ackerman drive[1] or holonomic drive[2], which the program is currently not designed to reproduce. No grippers, arms, or other objects can be added to or removed from the models as well; in particular, long arms add rotational inertia and move the center of gravity of the robot, aspects that cannot yet be simulated. As of the writing of this paper, only one robot can be run at a time, but this may be fixed by the time of the Global Conference.

Not all aspects of robot design are simulated at this point. Battery charge, one of the critical variables involved when performing repeated runs in the heat of competition, is not a factor in the code; the power_level() function or gc_batt_voltage variable returns a constant value, and motors/servos are unaffected by any real or virtual battery discharge rate. Errors can be introduced into tests by initial position, board construction, or human judgment, but the simulator does not cover these errors, possibly luring an uneducated user into a false sense of consistency. Arc turns in particular tend to be dependent on a variety of factors not covered in the code, so avoid relying on the current simulator's results to precisely calibrate high-speed arc turns.

The shortcut method of re-parsing C code to Java code also causes shortcomings over a real C parser. Some valid code, such as pointer and structure-heavy programs or extensive interchanging of boolean and integer variables, may fail to properly compile in the simulator even though it runs on a CBC or the KISS simulator. Botball programs tend to be sequential in nature, so this issue is somewhat acceptable; given more time, however, a real and standards- compliant C parser should be implemented instead. Programs that try to use files, Pthreads and mutexes [2], or OI scripting [3] tend to fail as well. While #define is fully supported, #include and #use are unreliable at best and #ifdef/#if does not work at all; a solution may be available by the time of this paper's publication.

3 Future Work

The graphics of this simulator, while improved, could be made flashier by incorporating 3D views. 3D modeling could add features such as climbing over walls and multi-level tables, and would also allow sensors to be aimed more realistically. A future simulator might allow simple modifications to be made to the models; this would allow scoring objects to be placed on the board and allow evaluation of test runs based on robot performance. While this simulator makes errors evident, it could do a lot more to ease error recovery. A

[1] *Ackerman drive*: the drive system used by most automobiles where the front wheels are steered and the back wheels are powered.

[2] *Holonomic drive*: a drive system where the wheels are not perpendicular to the ground, allowing for movement in any direction and rotation in place while driving.

measurement tool to find out how much the code needs to be changed would be useful, along with a replay button to explicitly review any section of the run at a modified time speed.

Support for a camera would be nice; blob tracking is not too difficult to code in Java, but accessing web cams is another story. Rendering the actual scene visible from the robot's point of view is even more difficult, which would probably involve a 3-D representation of the entire world and an improved modeling scheme. Camera code is probably best left to an actual test, as issues with brightness/contrast, color balance, frame rate, and sharpness contribute to the difficulties of good camera use. In addition, the sensor code needs tuning; a conditions button to test a variety of different settings for the game table would be great for reliability testing, and the sonar does not yet work. ET sensors need to be modified to change their results according to the type of object in view. Accelerometer support is also buggy, and every kind of touch sensor behaves identically.

4 Interested?

The simulator's current source code is available at http://code.google.com/p/botballsim. Follow the instructions if you want to look at the code; executable versions may be available by the time this paper is published. Keep in mind that this is development software which could be unstable or misleading; do not blame any failures at competition on this simulator or any other. The goal of Botball is still to have fun, and you can spend less time debugging and more time having real fun using this new simulator.

References

[1] National Institute of Standards and Technology, "USARSim: Unified System for Automation and Robot Simulation," *National Institutes of Standards and Technology*, 2007. [Online]. Available: http://usarsim.sourceforge.net/ [Accessed: September 16, 2009]

[2] Myers, Ethan, "Multiprocessing using Pthreads and Mutexes," presented at 2009 Global Conference for Educational Robotics, Leesburg, Virginia.

[3] Rand, Jeremy, "Advanced Create OI Scripting: CBC Motors and Sensors, Subroutines, Longer Scripts, and More – No CBC Required," presented at 2009 Global Conference for Educational Robotics, Leesburg, Virginia.