Published at:
International Conference on Autonomous Robots and Agents, ICARA 2004, Dec. 2004, Palmerston North, New Zealand, pp. 46-51 (6)

Mobile Robot Simulation with Realistic Error Models

Andreas Koestler, Thomas Bräunl
Mobile Robot Lab
The University of Western Australia, Perth
http://robotics.ee.uwa.edu.au {koestler, tb}@ee.uwa.edu.au

Abstract

EyeSim is a multi-robot, multi-tasking simulation system that allows realistic simulation of mobile robots with either differential drive, Ackermann steering, or omni-directional Mechanum wheel drive. The software development kit (SDK) for EyeSim is identical to the RoBIOS operating system for the real EyeBot robots, including a virtual camera system. A realistic error model for sensors and actuators allows the development, testing, and debuging of robust robot programs that can cope with real environments.

Keywords: mobile robot simulation, synthetic camera, error model, EyeBot, EyeSim

1 Introduction

We have developed a number of mobile robot simulation systems in the past, all with the goal to create an artificial robot environment - as close to reality as possible - as a testbed for mobile robot algorithms [1], [3]. What all previous simulators had in common with the "EyeSim" simulator presented here, is the duplication of a real robot's API (application programmer interface), the simulation of all its sensors and actuators, adjustable error models, and the generation of a virtual camera image that can be fed back into the application program.

This simulation system uses the API from the RoBIOS (Robot BIOS) operating system of the EyeBot project [2]. It has been successfully used in a number of projects involving time consuming experiments with Neural Networks, Genetic Algorithms, and Genetic Programming [4], [6].

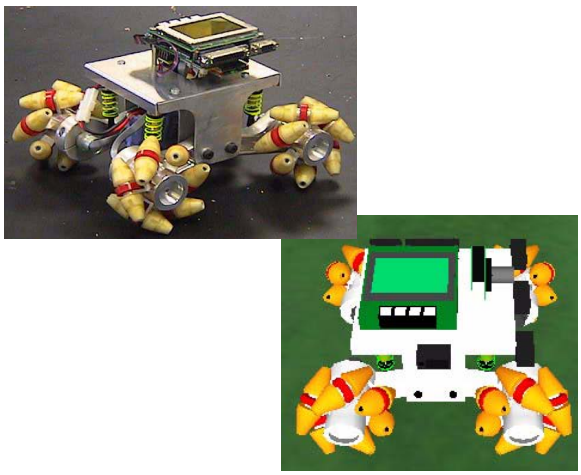

Figure 1: Real and simulated Omni robot

EyeSim implements both levels of robot locomotion available in the RoBIOS operating system:

(a) A high-level driving controller for driving straight and curve segments, as well as turning on the spot for differential-drive vehicles

(b) A low-level driving interface with direct simulation of motor actuators and shaft encoder feedback for vehicles with different drive mechanisms: differential drive, Ackermann steering, and omni-directional drive with Mecanum wheels (Figure 1).

The technique we used for implementing this simulation differs from most existing robot simulation systems [1],[12], which run the simulation as a separate program or process that communicates with the application by some message passing scheme. Instead, our simulation system dynamically links the application program at run-time and provides all API system functions for reading data from sensors and driving actuators. The simulation system is thread-based and uses the OpenGL, fltk and common-c++ libraries.

A number of parameter files determine the robot's physical dimensions, its performance and its modelled appearance. "Robi" files are used to describe each robot type. Environment files describe the shared driving scenery and can be specified either in "world" format or the simpler "maze" format. A "sim" file brings all these parts together for a simulation project. The sim file specifies the driving environment through the corresponding file name and lists each participating robot with its corresponding description file, starting pose, dynamic link library (the application program) and an optional graphics representation.

2 Drive Kinematics

The high-level driving interface implements driving functions for a v-ω (velocity - angular velocity) interface. These functions include VWDriveStraight, VWDriveCurve, VWDriveTurn, and VWDriveReady.

They include an implicit PID controller for velocity and position control.

The low-level interface allows direct manipulation of vehicle motors and reading of simulated quadrature encoder values. The robot description file determines the drive mode (differential drive, Ackermann drive, or omni drive), wheel base distance, wheel diameter, maximum wheel velocity and encoder ticks per wheel revolution. The description file also associates symbolic names for motors, servos (for Ackermann steering) and encoders, in a similar way as the HDT (hardware description file) does on the real EyeBot robot hardware.

The low-level forward kinematics for differential drive vehicles is defined is as follows:

$$\begin{bmatrix} v \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} \frac{c}{2} & \frac{c}{2} \\ -\frac{c}{d} & \frac{c}{d} \end{bmatrix} \begin{bmatrix} \dot{\theta}_L \\ \dot{\theta}_R \end{bmatrix}$$

- c is the factor relating ticks/s to m/s in wheel speed
- d is the distance between the two wheels

No formula is required for Ackermann steering, as both speed and driving angle are directly controlled by robot (or simulation) commands.

The low-level forward kinematics for omni-directional drive vehicles with four Mecanum wheels is defined is as follows:

$$\begin{bmatrix} v_x \\ v_y \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} \frac{c}{4} & \frac{c}{4} & \frac{c}{4} & \frac{c}{4} \\ -\frac{c}{4} & \frac{c}{4} & \frac{c}{4} & -\frac{c}{4} \\ -\frac{c}{4 \cdot d} & \frac{c}{4 \cdot d} & \frac{c}{4 \cdot d} & -\frac{c}{4 \cdot d} \end{bmatrix} \begin{bmatrix} \dot{\theta}_{FL} \\ \dot{\theta}_{FR} \\ \dot{\theta}_{BL} \\ \dot{\theta}_{BR} \end{bmatrix}$$

- c is the factor relating ticks/s to m/s in wheel speed
- d is the (l-r) distance between the two wheel pairs

3 Simulation Interface

The EyeSim user interface mainly consists of two parts: The 3D visualization of the world containing the robots, objects and environment, and the control interface, through which the user can change simulation parameters and thereby interact with the simulation. For the 3D rendering, a simple yet effective 3D engine based on OpenGL has been developed. It is capable to load and visualize models created with

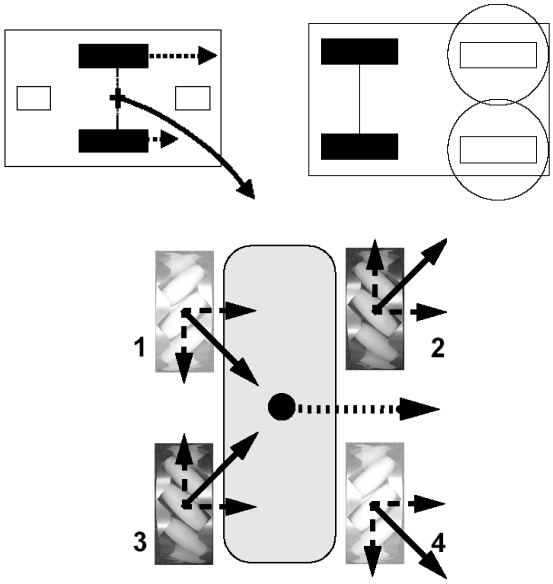

Figure 2: Low-level driving models: Differential, Ackermann, Omni-directional

Milkshape3D [11], a shareware polygon modelling tool, and it allows the user to explore the created world by freely rotating and translating the scene camera. Another advantage of having a 3D representation of the simulated world is, that it greatly simplifies debugging as the user has visual perception of the world and the robots interacting with each other. The control interface, created with the fltk library [7], gives access to simulation parameters, such as the simulation speed, the error model or the visualization parameters of the sensors and it also allows the user to manually change the behavior of the simulation by changing the position and rotation values of the robots in the environment. Another important feature is the simulation of the full functionality of the EyeBots liquid crystal display (LCD) including its control buttons and thereby allowing to control the simulation application as it would be possible on the real robot.

4 Sensor Modeling

As the perception of a robot's environment, simulated or real, is performed by its sensors, it was not very surprising that some effort was put into appropriately modelling these sensors. The EyeBot can be equipped with a camera, representing the vision sensor, and an arbitrary number of position sensitive devices (PSD).

4.1 Sensor Data Calculation

Probably one of the most outstanding features of EyeSim is that it simulates the robot's camera, hence fulfilling the requirement for writing programs that use real-time image processing. The image is obtained by

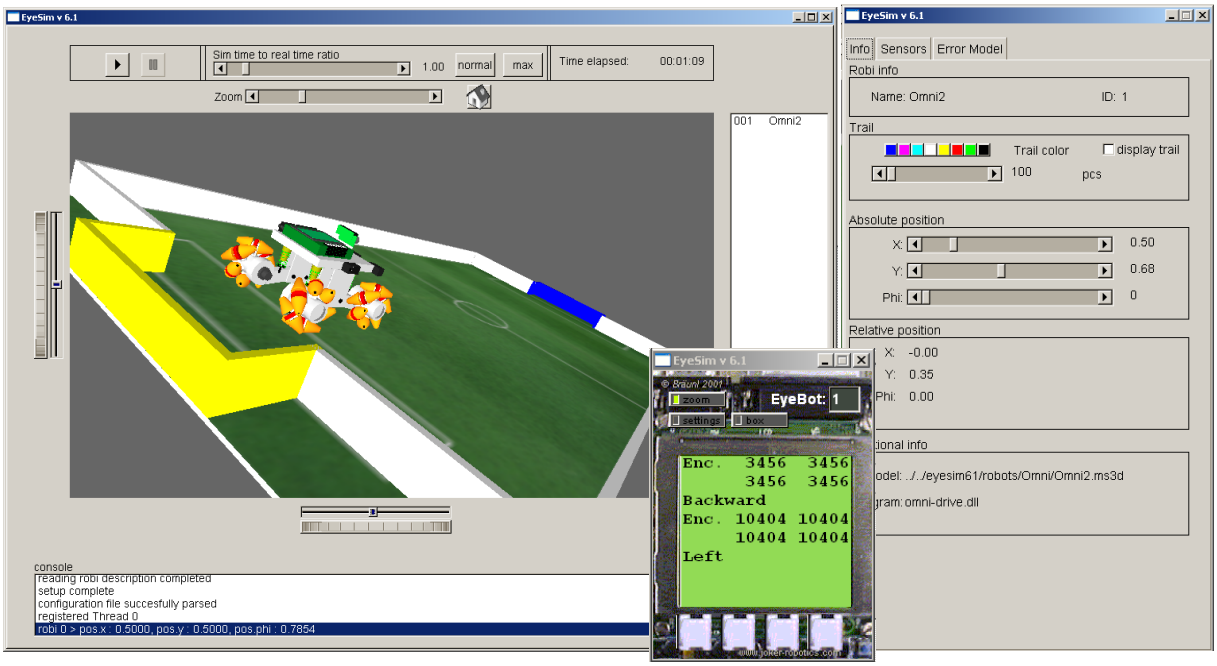

Figure 3: User Interface

rendering the scene in the robot's camera field of view to an offline buffer. It is accessible by the RoBIOS application programming interface using the CAMGetColFrame and CAMGetFrame functions for obtaining color and grayscale images, respectively.

The PSD sensors are implemented by casting a ray of the length representing the maximum range of the PSD in the sensor's direction, which is specified in the robot's description file. If the ray collides with an obstacle, such as another robot, a wall or an object placed in the environment, the distance to the object under consideration is calculated and stored as the sensor's distance measurement. Either continuous or single measurements can be initiated via the PSDStart command and the distance in mm can be obtained using the PSDGet function.

4.2 Sensor Data Visualization

EyeSim was developed to assist the user with debugging his or her applications. Therefore, it offers a built-in debugging mode offering a verbose output of its initialization and execution process. Thus errors in the simulator configuration can easily be determined. In addition, EyeSim offers a range of tools for simplifying debugging programs running on the robot by visualizing the sensor reading values, either graphically or numerically. The PSD sensors are visualized by drawing a line representing the infrared beam of the physical sensor device and the measured distance values are stored for further examination. It is also possible to visualize the camera's field of view by drawing the outlines of its view frustum. Sending and receiving radio messages is also graphically illustrated in the simulation system.

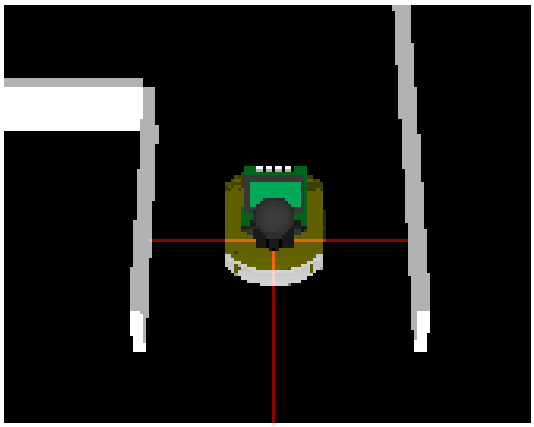

Figure 4: Sensor visalization

5 Error Models

The signals produced by the modeled sensors generally do not reflect the real world as they are calculated and synthetically created. Real world signals usually contain variances from the ideal signal, which we will refer to as noise. Noise may be caused by a wide range of sources, e.g. variations of detector sensitivity, environmental variations, transmission errors or quantization errors. The overall performance of a sensor is ultimately limited by the noise that is added to the signal. Therefore, introducing noise to the simulation system is crucial to test, evaluate and improve the robustness of the developed algorithms and procedures and to increase their performance. This is especially useful for the development of vision based

applications. One major issue is that noise typically is of a random nature or dependent on device characteristics and its distribution is therefore unknown or hard to calculate. However, statistical models exist which are easy to implement and yet reflect the nature of noise.

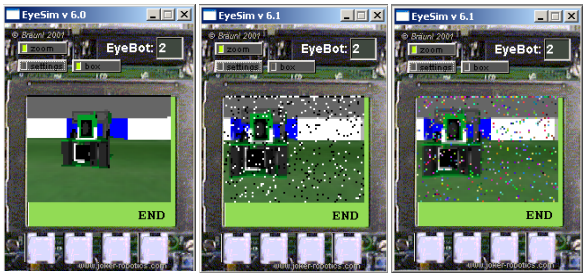

Figure 5: Original image, salt&pepper noise, 100s&1000s noise

For EyeSim we implemented the following additive error model for the PSD sensors, the vision sensor and the v-ω driving interface.

$$f(i) = s(i) + n(i)$$

f is the resulting signal

s is the undistorted source signal

n is the white noise, modelled by a normally distributed, zero mean (Gaussian) random process

For the vision system we additionally modeled a source of impulse noise given by

$$f(i) = (1-e)s(i) + en(i)$$

with

$$e = \begin{cases} 1, \text{ with probability } P_e \\ 0, \text{ otherwise} \end{cases}$$

where n(i) now is either randomly $\hat{s}(i)$ (maximum grayscale value for white) or 0 (black) hence the name *"salt and pepper noise"*, or a random color - in that case called *"hundreds and thousands"* noise.

5.1 Error Models for Wireless Communication

There are many reasons for noise in transmission systems: E.g. random noise, intermodulation noise and crosstalk. The sum of this sources of noise is called channel noise which is modeled by swapping random bits and dropping whole communication packages. The channel model used in EyeSim is the binary symmetric channel model with a specified bit error rate P_b. Additionaly the probability P_p of dropping a whole communication package can be specified. This allows the evaluation of the used communications protocol error stability.

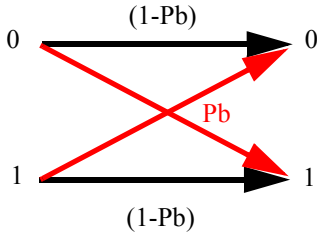

Figure 6: Binary Symmetric Channel

6 Time Model

As we described in Chapter 2, the robots' movement can be depicted by specifying an angular and linear velocity. Therefore it was necessary to implement a proper time model to keep track of the robots trajectory by querying its position at a certain time. Another reason for modelling time is to provide fairness in cooperative or competitive applications. According to our simulation system we can distinguish three types of time.

6.1 Local and global timekeeping

Virtual time

The virtual time reflects the robots "life time". It is accumulated by adding the execution time values, measured on the Eyebot and stored in a lookup table, to the current virtual time. Therefore it coarsely depicts the execution time of the program on the Eyebot itself. As most of the commands used in an Eyebot program are RoBIOS application programming interface calls, and the execution time of native C calls is short in comparison to RoBIOS calls, native C calls are neglected as contributors to the virtual time.

Simulation time versus global time

The simulation time advances in discrete time steps (e.g. T=0.05s in the current simulation system). The time step T determines the rate the state of the world (including robot positions) gets updated. A time slot, ts(virtual time), can be assigned to a robots virtual time, where I is the largest integer with $IT \leq$ virtual time

Real time

Real time is the time passing in the real world since execution of the program on the host computer. As it reflects the real system time, it is not computed but obtained from the system clock. To run the simulation as fast as possible the RoBIOS calls delay times are ignored.

6.2 Synchronization

One reason for keeping all the robots synchronized is, that the robots can interact with each other. Either by stimulating their sensors, e.g. one robot is intersecting another robot's camera field of view, or by establishing a communication via the radio interface. In both cases it is absolutely crucial to keep the robots synchronized, otherwise messages from the future (relative to the global time) may occur, or a robot might records the image of another robot at a position where it will be in the future.

To achieve this task, each robot stores a time stamp for its last synchronization. Every time it calls a RoBIOS function, the current virtual time is computed and compared with the last synchronization time. If the robots virtual time does not lie within the global time slot, it sends a synchronization message to the core, gets suspended and waits for a wake up call. By suspending the core for a certain amount of time each iteration, it was also possible to implement time lapse and slow motion effects, however intermediate time steps have to be introduced in order to provide a fluent display update

7 Multithreading

RoBIOS is a multitasking operating system and supports both preemptive and cooperative multitasking. After the multitasking system has been specified and the threads have been initialized, they are added to the thread queue, providing the function address, a name, the priority, the stack size and a user id. The next step is to initiate the task switching: For cooperative multitasking this is done by the *reschedule* command, for preemptive multitasking this is done by the *permit* command. Threads can be terminated by themselves or other threads, suspended and resumed by any thread, or suspend themselves via a sleep command. Semaphores are available for secure inter-thread communication.

8 Related Work

To the best of our knowledge, we developed the first mobile robot simulation system employing synthetic vision as a feedback sensor in 1996 [1]. This system was limited to a single robot with vision system and required the use of special rendering hardware. Our new system presented here has been implemented completely in software and has no restrictions in the number of robots with vision systems.

Most other mobile robot simulation systems, e.g. Saphira [9] or 3D7 [12] are limited to simpler sensors such as sonar, infrared or laser sensors, but cannot deal with vision systems. Matsumoto et al. [10] implemented a vision simulation system very similar to our earlier system [1] in 1999.

Ho and Stark [8] examined the performance of a mobile robot system using vision by simulating the visual redundancy in the image. However, they did not use a fully computer-generated image nor simulate the complete mobile robot. A simulation system for a single vision-guided underwater vehicle has been described by Deltheil et al. [5]. Wang et al. [13] describe the Webots simulation environment for the Khepera mobile robot, however with only limited 1D or 2D vision capabilities.

9 Summary and Outlook

We have presented the EyeSim simulation system, which accurately models the RoBIOS API with realistic driving operations and adjustable, realistic error models. The EyeSim software package is available via Internet as public domain software:

```
robotics.ee.uwa.edu.au/eyebot/doc/sim/
    sim.html
robotics.ee.uwa.edu.au/eyebot/ftp/
```

10 References

[1] T. Bräunl, H. Stolz, "Mobile Robot Simulation with Sonar Sensors and Cameras", *Simulation*, vol. 69, no. 5, Nov. 1997, pp. 277-282 (6)

[2] T. Bräunl, "Research Relevance of Mobile Robot Competitions", *IEEE Robotics and Automation Magazine*, Dec. 1999, pp. (10)

[3] T. Bräunl, "Multi-Robot Simulation with 3D Image Generation", *IROS 2001*, Maui, pp. (6)

[4] T. Bräunl, *Embedded Robotics - Mobile Robot Design and Applications with Embedded Systems*, Springer 2003

[5] C. Deltheil, L. Didier E. Hospital, D. Brutzman, "Simulating an optical guidance system for the recovery of an unmanned underwater vehicle", *IEEE Journal of Oceanic Engineering*, vol. 25, no. 4 , Oct. 2000, pp 568 -574

[6] J. Du, T. Bräunl, "Collaborative Cube Clustering with Local Image Processing", *Proc. of the 2nd Intl. Symposium on Autonomous Minirobots for Research and Edutainment*, AMiRE 2003, Brisbane, Feb. 2003, pp. 247-248 (2)

[7] Fast Light Tool Kit, online, http://www.fltk.org/

[8] Y. Ho, L. Startk, "Top-Down Imapge Processing and Supervisory Control Limitations in Robotics: A Simulation Study", *ICAR'97*, Monterey, 1997, pp. 993-938

[9] K. Konolige, *Saphira Version 6.2 Manual*, [originally: Internal Report, SRI, Stanford, 1998] 2001, http://www.ai.sri.com/~konolige/saphira/

[10] Y. Matsumoto, T. Miyazaki, M. Inaba, H. Inoue, "View Simulation System: A Mobile Robot Simulator using VR Technology", *Proc. Intl. Conf. on Intelligent Robots and Systems*, IEEE/RSJ 1999, pp. 936-941

[11] Milkshape 3D, online, http://www.milkshape3d.com/

[12] R. Trieb, *Simulation as a tool for design and optimization of autonomous mobile robots* (in German), Ph.D. thesis, Univ. Kaiserslautern, 1996

[13] L. Wang, K. Tan, V. Prahlad, "Developing Khepera Robot Applications in a Webots Environment", 2000 *Intl. Symposium on Micromechatronics and Human Science*, IEEE, 2000, pp. 71-76