Computer Science Department of The University of Auckland
CITR at Tamaki Campus (http://www.citr.auckland.ac.nz)

CITR-TR-58 March 2000

The EyeSim Mobile Robot Simulator

Thomas Bräunl[1]

Abstract

EyeSim is a 2D simulator for the EyeBot mobile robot systems. The simulator is implemented as a library, which is linked to the robot application program. It allows the concurrent simulation of multiple robots in the same environment. The simulator integrates the robot console with buttons and LCD; it implements a "v-omega" (linear and angular velocity) driving interface and models a variety of sensors: shaft encoders, bumpers, infra-red proximity sensors, infra-red position sensors, and interfaces to a live color camera. The simulation can be executed in a "perfect" environment or with certain error ranges for individual sensors.

The University of Western Australia
Centre for Intelligent Information Proc. Sys (CIIPS)
Department of Electrical and Electronic Engineering
Perth, Western Australia

The EyeSim Mobile Robot Simulator

Thomas Bräunl

The University of Western Australia
Centre for Intelligent Information Proc. Sys. (CIIPS)
Department of Electrical and Electronic Engineering
Perth, Western Australia

visiting:
The University of Auckland
Centre for Image Technology and Robotics (CITR)
Department of Computer Science
Auckland, New Zealand

Abstract

EyeSim is a 2D simulator for the *EyeBot* mobile robot systems. The simulator is implemented as a library, which is linked to the robot application program. It allows the concurrent simulation of multiple robots in the same environment. The simulator integrates the robot console with buttons and LCD; it implements a "v-omega" (linear and angular velocity) driving interface and models a variety of sensors: shaft encoders, bumpers, infra-red proximity sensors, infra-red position sensors, and interfaces to a live color camera. The simulation can be executed in a "perfect" environment or with certain error ranges for individual sensors.

1. System Concept

The goal of this project was to develop a tool, which allows the construction of simulated robot environments and the testing of mobile robot programs before they are used on the real vehicles. A simulation environment like this is especially useful for the programming techniques of neural networks and genetic algorithms. These have to gather large numbers of test or training data, which is very difficult to obtain from the actual vehicles. Potentially harmful collisions, e.g. due to untrained neural networks or errors, do not cause any harm in the simulation environment.

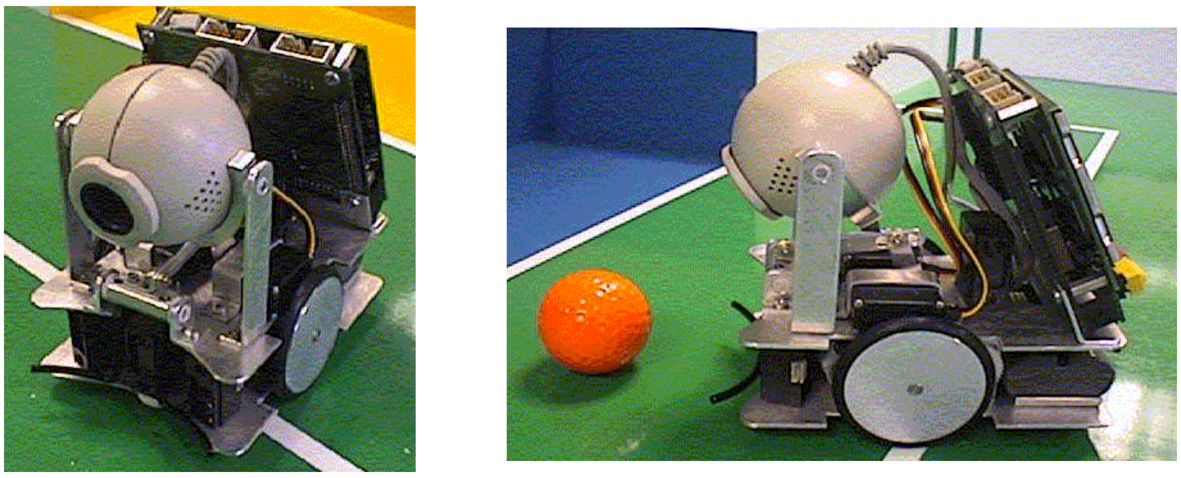


Figure 1: EyeBot mobile robot

The technique we used for implementing this simulation differs considerably from most existing robot simulation systems [1],[5]. While most other simulation systems run the simulation

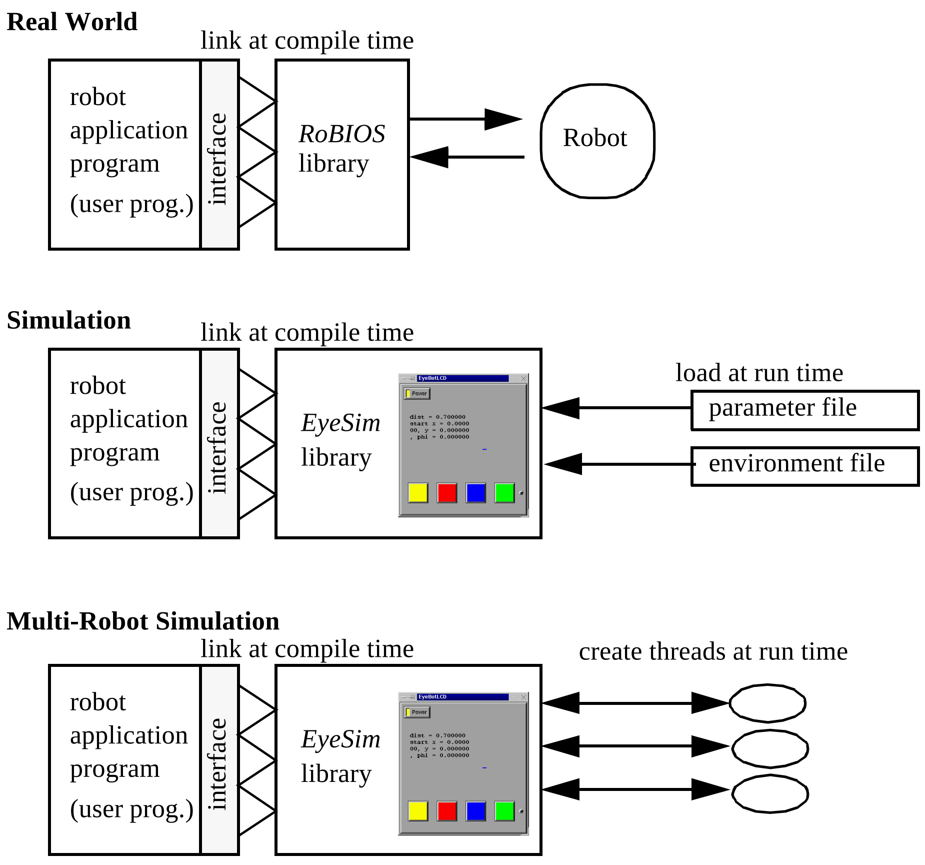


Figure 2: System concept

as a separate program or process, which communicates with the application by some message passing scheme, we implemented the whole simulator as a collection of system functions, which are liked to the application program and are called like system subroutines.

As shown in Figure 2 (top), a robot application program is compiled and linked to the *RoBIOS* library, in order to be executed on a real robot system. The application program makes system calls to individual *RoBIOS* library functions, e.g. for activating driving motors or reading sensor input. In the simulation scenario, shown in Figure 2 (middle), the compiled program is now linked to the *EyeSim* simulation library instead of the *RoBIOS* library. This simulation library implements exactly the same functions, but also activates the screen displays of robot console and robot driving environment. Two additional files are loaded at run time, a parameter file and an environment file. The parameter file specifies:

- Robot name
- Robot size
- Robot maximum speed
- Robot sensors with their position and orientation relative to the robot's center
  - Infra-red distance sensors
  - Infra-red proximity sensors
  - Tactile bumpers

- Error margins for sensors and actuators
- Robot start position and orientation
- Number of robots in case of multi-robot simulation
- Name of environment file

The environment file can be of either world format or maze format and specifies the robot's environment as a 2D graph. Environments are discussed in more detail below.

In case of a multi-robot simulation, Figure 2 (bottom), nothing changes in the compilation and linking process to the single-robot simulation. However, during run-time, individual threads are created to simulate each of the robots concurrently. Each thread executes the same robot program individually on local data, but all threads share the common environment through which they interact. Executing the same program on all robots is not a restriction. Since each robot has a unique id-number, they can easily branch to different subroutines and thus effectively execute different programs.

2. Sensor - Actuator Modeling

The *EyeBot* robot has two motors to maneuver it in differential steering and is equipped with a variety of sensors. These are:

- Shaft encoders
- Tactile bumpers
- Infra-red proximity sensors
- Infra-red distance measurement sensors
- Digital grayscale or color camera

On the real robot, the *RoBIOS* operating system library provides access routines for each of the sensor types. For the *EyeSim* simulator, exactly the same interface has been kept but with different implementations to evaluate a particular sensor in a simulated environment as opposed to the actual device drivers.

The shaft encoders give driving feedback on a very low level and are rarely used directly. The same holds for the low level motor drivers, which allow the setting of a voltage level to an individual motor. Instead, application programs use a v-omega interface, which allows specifying a translational and rotational vehicle speed. The v-omega library includes a PID controller and handles speed ramps. So the shaft encoder feedback is used implicitly and not retuned to the user program. There are, however, high level routines in the v-omega interface returning vehicle position and orientation relative to its starting point.

The simulation library, therefore, does not contain any of the low-level motor and shaft encoder routines. Instead, all functions of the v-omega interface have been re-implemented to constitute a driving simulator. E.g. function "VWInit" will open a second window, displaying the robots in their simulated environment. A 2D graphics representation is shown with all sensor readings being displayed below. The user can pause/continue the simulation and pick/ move individual robots. Interaction between robots and the shared environment is via "VWDrive" function calls. Whenever a robot user program calls on of these v-omega routines, the corresponding values are set in the simulation and a periodically called update process computes all new robot positions and sensor values and updates the window display. The routines for all other sensors simply return values stored in the central simulation environment, which are also periodically re-calculated by the update process.

The bumper sensor is simulated by computing intersections between the robot (modeled as a circle) with any obstacle (modeled as line segments) in the environment or another robot. Bump sensors can be configured as a certain sector range around the robot. That way several bumpers can be used to determine the contact position.

The *EyeBot* uses two different kinds of infra-red sensors. One is a binary sensor ("proxy") which is activated if the distance to an obstacle is below a certain threshold, the other is a position sensitive device ("PSD"), which returns a distance value to the nearest obstacle. Sensors can be freely positioned and orientated around a robot as specified in the robot parameter file. This allows testing and comparing of the performance of different sensor placings. For the simulation process, the distances of all infra-red sensors at their current positions and orientations towards the nearest obstacles are determined.

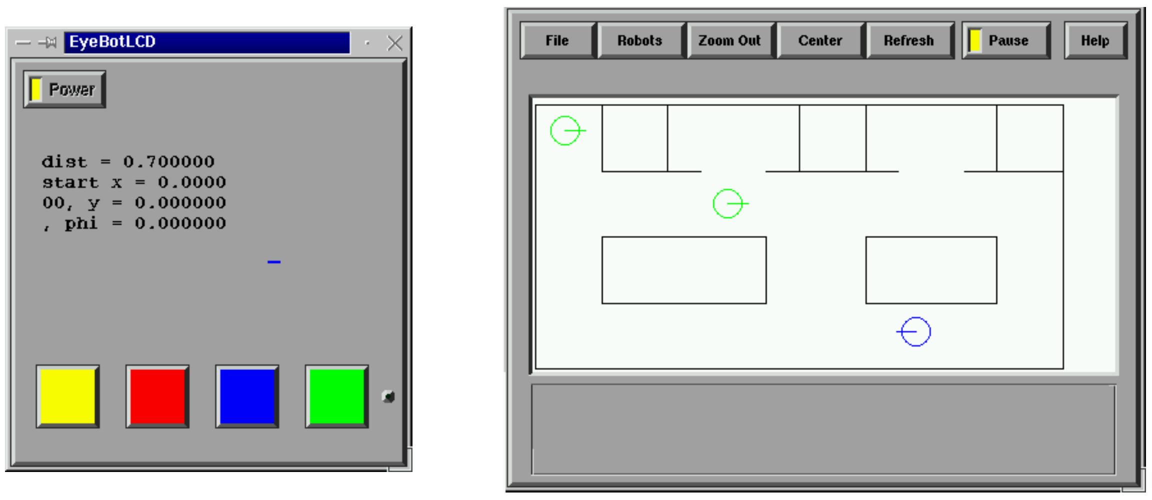
For the digital camera, it seemed more appropriate and more realistic to allow the input of a real camera instead of generating a virtual camera image. The video input routines return an image from the digital camera connected to the workstation. This way, typical applications involving on-board vision can be tested, by "driving a camera by hand" through the robot environment. However, an extension planned for a future version of *EyeSim* will use a variation of the "OpenInventor" library [7] to generate a computer-rendered camera image instead of using an actual camera.

3. Simulation Interface

The *EyeSim* interface is split into two parts, a robot panel and a driving environment (Figure 3). The robot panel is a replication of each robot’s LCD and input buttons. This allows a direct interaction with the robot by pressing buttons and displaying status messages, data values or graphics on the screen. Each robot has an individual panel, while the all share the common driving environment. This is a bird’s eye view (2D) of the robot driving area, with each robot displayed as a circle with a marker indicating the robot’s front.

Simulation execution can be halted by pressing the "Pause" button and robots can be relocated or turned by clicking and dragging. Zoom buttons allow to change the view of a larger environment, while another button centers the display around the first (or only) robot.

Since the whole simulator is implemented as a function library, which is called in lieu of standard *RoBIOS* functions, the simulator can only be started by a robot application program when *RoBIOS/EyeSim* functions are called. The first call to any of these library functions will initialize and activate the panel display. All subsequent text or graphics outputs will be displayed in this window. The first driving or sensor related library function call (e.g. driving initialization or infra-red sensor initialization) will then initialize and display the driving environment window and load the simulated robot environment from a file.



```
dist = 0.700000
start x = 0.0000
00, y = 0.000000
, phi = 0.000000
```

Figure 3: Robot panel and driving environment

4. Robot Environments

All environments are modeled by 2D line segments and can be loaded from text files. Possible formats are either the *world format* used in the "Saphira" robot operating system [4] or the *maze format* developed by us following the well known Micro Mouse Contest notation [2].

4.1 World Format

The environment is described by a text file. It specifies walls as straight line segments by their start and end point with dimensions in millimeters. An implicit stack allows the specification of a substructure in local coordinates, without having to translate and rotate line segments. Comments are allowed following a semicolon until the end of a line.

The world format starts by specifying the total world size in mm, e.g.

```
width  4680
height 3240
```

Wall segments are specified as 2D lines [x1,y1, x2,y2], so four integers are required for each line, e.g.

```
;rectangle
0 0 0 1440
0 0 2880 0
0 1440 2880 1440
2880 0 2880 1440
```

Through an implicit stack, local poses (position & orientation) can be set. This allows an easier description of an object in object coordinates, which may be offset and rotated in world coordinates. To do so, the definition of an object (a collection of line segments) is being enclosed within a push and a pop statement, which may be nested. Push requires the pose parameters [x,y, phi], while pop does not have any parameters, e.g.

```
;two lines translated to [100,100] and rotated by 45 degrees
push 100 100 45
0 0 200   0
0 0 200 200
pop
```

The starting position and orientation of a robot may be specified by its pose [x,y, phi], e.g.

```
position 180 1260 -90
```

4.2 Maze Format

The Maze format is a very simple input format for environments with right angles only, such as the Micro Mouse competitions. We wanted the simulator to be able to read typical "natural" graphics ASCII maze representations, which are available from the web, like the one below:

```
 _________________
|  _________|     |
| |  _____  | |___|
| | |_____  | | | |
| |  _  __|___|  _|
| |_|____________ |
| |___    |  _  | |
|  _  | |___| | __|
| | | |   | ____  |
|S|_____|_______|_|
```

Each wall is specified by single characters within a line. A "|" (at odd positions in a line, 1,3,5,..) denotes a wall segment in y-direction, a "_" (at even positions in a line, 2,4,6,..) is a wall segment in x-direction. So, each line contains in fact the horizontal walls of its coordinate and the vertical wall segments of the line above it.

This following example defines a rectangle with two dividing walls:

```
 _ _ _
|    _|
|_|_ _|
```

The following notation is identical, but avoids gaps and therefore looks nicer:

```
 _____
|    _|
|_|___|
```

Extra characters may be added to a maze to indicate starting positions of one or multiple robots. Uppercase characters assume a wall below the character, while lowercase letters do not. The letters U (or S), D, L, R may be used at any position, indication a start orientation upwards, down, left, or right. In the last line of the maze file, the size of a wall segment can be specified in mm (default value 360 mm) as a single integer number.

A soccer ball can be inserted by using the symbol "o". The robots can then interact with the ball by pushing or kicking it, e.g.

```
   _____________________________________________________
  |                                                     |
  |                                                     |
  |          r                               l          |
  |                                                     |
 _|                                                     |_
|            r                               l            |
|                                                         |
|  r                         o                         l  |
|                                                         |
|_           r                               l           _|
  |                                                     |
  |                                                     |
  |          r                               l          |
  |                                                     |
  |_____________________________________________________|
100
```

5. Multiple Robots

A multi-robot simulation can be initiated in two ways. The first method is starting a simulation as a single robot simulation, then the user selects item "Add robot" from the "Robots" menu and interactively places the new robot in the environment. For the second method the associated parameter file specifies a number of robots with their respective starting poses.

5.1 Thread Creation

Individual threads for each robot are created at run-time, either when a new robot is being added (method 1) or for the whole set of robots (method 2). Posix threads and semaphores are used for synchronization of the robots [6],[3], which each robot receiving a unique id-number upon creation of its thread.

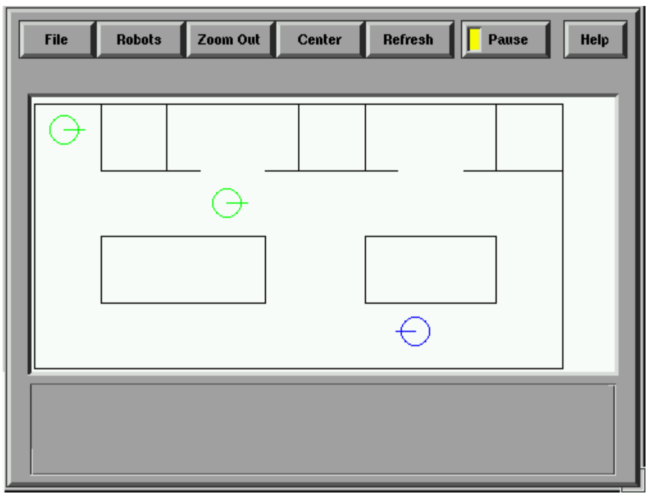


Figure 4: Multi-robot environment

Since there is only one robot application program, which has been linked to the simulation library, all robots threads which are new initialized will start to execute this very same program. However, this is not a restriction since the program can arbitrarily branch according to its id-number, e.g. for the different roles in a robot soccer team:

```
void main()
{ switch (OSMachineID())
  { case ’1’: goalkeeper(); break;
    case ’2’:
    case ’3’: defender(); break;
    case ’4’:
    case ’5’: striker(); break;
  }
}
```

A complex task is executing all parallel threads up to the same point, while also avoiding a recursive chain of new robot thread creations. Only the first robot thread may create new robots. If multiple robots are defined in the parameter file, this thread will then wait at the end of its initialization until all other robot threads have been created, commenced their execution, and reached the same initialization subroutine call. This subroutine call is the first call of a robot-related *RoBIOS/EyeSim* function, e.g. "VWInit".

5.2 Robot Interaction

A single robot interacts with its environment via its sensors and actuators (v-omega-driving interface). Since the environment is in 2D, each line represents an obstacle of unspecified height. Each of the robot’s distance sensors measures the free space to the nearest obstacle in its orientation and then returns an appropriate signal value. This does not necessarily mean that a sensor will return the correct distance value. E.g. the infra-red sensors only work within a certain range. If a distance is above or below this range, the sensor will return out-of-bounds values and the same behavior has been modeled for the simulation. If a robot drives into an obstacle, it will be stopped and a warning message will be displayed in the environment window.

Whenever several robots interact in an environment, their respective threads are executed concurrently. All robot position changes (through driving) are being made by calling a library function of the v-omega-driving interface and can therefore be used to update the common environment. All subsequent sensor readings are similar library calls, which now take into account the updated robot positions (e.g. when one robot is detected as an obstacle by another robot’s sensor). Collisions between robots are also detected and reported in the environment window. Robots are displayed in different colors to facilitate distinguishing between them. Also, trail markings can be activated to allow visualization of the path that each robot has travelled.

Summary and Outlook

We have introduced a new model of a simulation system. Instead of a simulation process which communicates with an application program process, our simulation system is a library of system functions, which is linked to the application after the compilation step. This eliminates the need for communication primitives, since all simulation operations are now triggered by procedure calls. The first call to any simulation function will initialize the simulator and open a controller window (LCD and buttons), the first call to a sensor ar actuator function will initialize the simulation environment and open a window showing a 2D view of the robot in the selected environment.

The simulation system can deal with multiple robots in the same environment. The number of robots and their respective positions and orientations can be either set in a configuration file or interactively by picking and placing directly in the environment window, once the first robot has been initialized.

The simulator includes an error model, which allows to run a simulation either in a "perfect" world, or it allows the setting of an error rate in both actuators and sensors, to allow for a more realistic simulation. This can also be used for making a robot application program more robust.

Currently, we are planning to extend the simulation system into a 3D system by integrating "OpenInventor" functions as a virtual camera system, similar to the Mobs robot simulation system [1]. This will enable us to replace the physical camera connection by a computer generated rendered image, and - most important - relax the single camera restriction. Eventually, we will be able to simulate any number of robots concurrently in a shared environment, each with its local sensors including a simulated camera.

The *EyeSim* software package is available via Internet as public domain software:

```
http://www.ee.uwa.edu.au/~braunl/eyebot/sim/sim.html
http://www.ee.uwa.edu.au/~braunl/eyebot/ftp/
```

Acknowledgments

The author would like to thank Nicholas Tay, Elliot Nichols, Ghee Ong, and Min-Tyi Chao for contributing to the implementation of *EyeSim*, as well as Reinhard Klette and Jacky Baltes for the invitation to do joint research at CITR at the University of Auckland, during which this paper has been written.

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] IEEE, *POSIX Threads Standard, 1997*

[4] K. Konolige, *Saphira Version 6.1 Manual*, Internal Report, SRI, Stanford, 1998, http://www.ai.sri.com/~konolige/saphira/

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

[6] T. Wagner, D. Towsley, *Getting Started with POSIX Threads*, internal report, Dept. of Computer Science, Univ. of Massachusetts at Amherst, July 1995

[7] J. Wernecke, *The Inventor Mentor*, Addison Wesley, Reading MA, 1994