SPREAD: A DISTRIBUTED SIMULATION TOOLKIT

P.Chernett[1], V. Callaghan[1], C. Ching[2], D. Diemmi[3]

email vic@essex.ac.uk

Abstract

This article describes "SPREAD", a simulation tool kit and its use in building "Virtual Robots", a simulation of multiple mobile robot vehicles used in the teaching of Computer Science at university level. A novel aspect of the simulator is the use of PVM [1] to achieve high performance at low cost by using spare CPU cycles on large numbers of networked workstations.

Acknowledgments

Many people have been involved in developing the simulator. In particular our thanks go to Paul Marriot who wrote the very first version as long ago as 1992, to Caleb Ying who was responsible for many of the architectural decisions, and to Christos Voudouris for many ideas and clarifications that have found their way into the programs.

1. Introduction:

The computer science department at the University of Essex (in common with other universities contributing to this issue) has for some time used simple robot vehicles, which we characterize as Intelligent Autonomous Vehicles (IAVs), in the teaching of undergraduate computer scientists This work is described in more detail elsewhere in this issue.
A victim of its success, our robotics facility has soon reached capacity before the needs of all students have been catered for. One way to provide wider access to limited and expensive physical resources such as robots is by simulation. A prime example of this is in the area of multi-agent AI where simulation can allow experimentation with large numbers of virtual robots where only a few may be available in reality.

2. Other Related Systems

There are, of course, several commercially available packages for supporting this type of simulation. Unfortunately the cost of these, and the specialist workstations needed to run them put them beyond the means of departments that only require the facility as a relatively small part of their overall activity. There are several low-cost or shareware simulation packages available but many (for example Simderella [2]) assume static robots with manipulator arms rather than IAVs. Other popular robot vehicle simulators, such as Xmouse [3], are usually built on very simple models that have little relation to any real robots and can often only deal with single robots. Those that do simulate multiple robots such as Mission Lab [4] are often at a

[1] University of Essex, England

[2] Motorola Inc.

[3] University of Parma, Italy

very high level and don't allow detailed simulations of individual robots. Yet others such as Khepera [5] are tied to particular vehicles.

3. Overview

Our approach has thus been to develop the SPREAD simulation engine to supports the simulation of complex worlds, inhabited by multiple autonomous vehicles, each of which may be modeled as many parallel embedded processes. Overall performance is maintained by distributing execution across a network of computers. An important design aim was machine independence so that the system could be used in a wide range of institutions while still taking advantage of whatever hardware they happened to have available. This portability extends to the worlds to be simulated as well as to the simulator itself.

In this paper SPREAD is discussed in the context of the "Virtual Robots" simulation tool used in the Brooker Laboratory for Intelligent Embedded Systems at Essex (whose activities are described in greater detail elsewhere in this issue). Virtual Robots was built using the SPREAD toolkit and has been in use in the department for about 2 years.

We will first describe the architecture of SPREAD and its use in Virtual Robots. This will be followed by a case study showing how the system was used by a student to design a new sensor, some performance figures, and finally a brief description of our future plans for the simulator.

4. The SPREAD Architecture

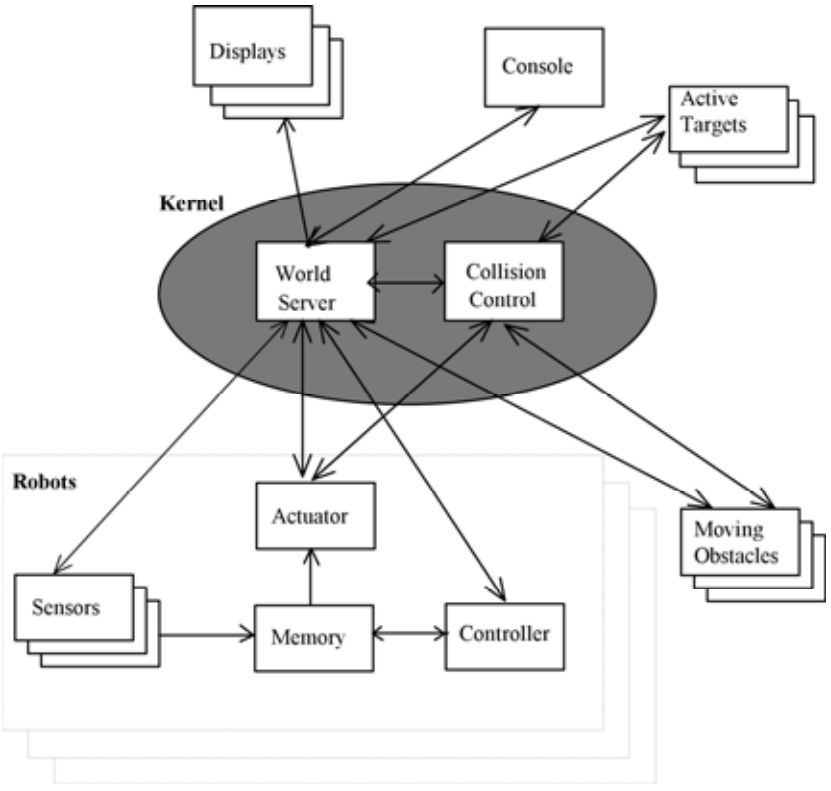

Figure 4-1 Overall architecture of the simulator

The simulator structure (see Figure 4-1) consists of the following types of module: console, display, world, collision control, actuator, sensor, memory, controller, moving obstacle and active target.

A single world server and collision control server form the kernel of the simulator. The world server maintains a database of the position of all objects in the simulated world. The collision control module negotiates with objects that have collided and reports back the result to the world server.

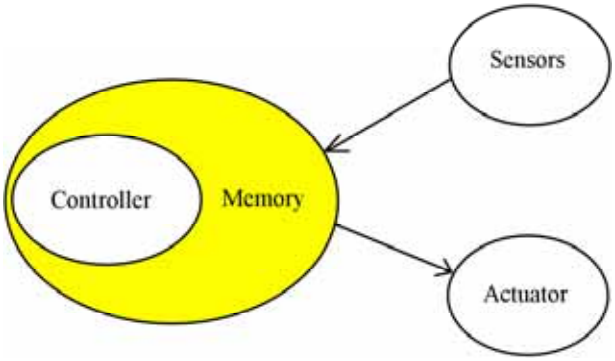

Figure 4-2 General Architecture of a Mobile Robot

Each robot (See Figure 4-2) consists of a controller, an associated set of sensors and, if the I/O to be modeled is memory mapped, a memory module. The console, of which there must be exactly one, provides control of the simulation itself and is responsible for starting and terminating the simulation as well as making run-time adjustments such as the level of detail to be simulated. Display modules, of which there may be several, present the current state of the simulation to the users

Along with the modules of the simulation itself the complete Virtual Robots package includes graphical editors for designing environments and setting up initial conditions for simulator runs, such as the position and orientation of the IAVs.

5. The user interface

There are four classes of SPREAD user:

- those that simply use it via an GUI like Virtual Robots, together with an accumulated set of robot and robot parts,
- those that want to write and test robot controllers, normally in parallel with testing the same controllers on real robots,
- those that want to add new simulations of robot parts, and finally
- those that want to build new GUIs on top of SPREAD

The way each of these groups interacts with SPREAD is discussed below:

5.1 Using an existing GUI

The only reasonably complete GUI to be written so far is the "Virtual Robots" simulator that we use at Essex although a start has been made on an X version. Virtual Robots is described in more detail below.

There are three tasks that the GUI has to perform:

First the initial conditions for a simulator run have to be set up. This includes placing all the static objects into the environment, including boundary walls, obstacles, detectable tracks on the floor and so on. Active targets such as beacons need to be placed and have their characteristics defined such as aperture angles, ranges and identification numbers. Robots need to be "built" by placing sensors and actuators and deciding their characteristics. Finally the initial position and orientation of the robots need to be decided. In a fully complete system all of this should be achievable graphically, rather like using a drawing program.

The second task of the GUI is to start up the display processes. There can be as many of these as desired and they can be placed on any of the participating workstations. Virtual Robots has only one sort of display that is described below.

Finally provision must be made for controlling the simulation itself. There can only be one of these and in Virtual Robots it comprises a simple panel with "START", "Stop", "Reset" and "Exit" buttons. In addition it has a slider that sets the amount of real time that each "tick" of simulated time represents.

5.2 Writing new robot controllers

Assuming that a simulated robot has been set up that matches the real robot under test the SPREAD API imposes very few changes on the C or C++ source code in which controllers are normally written. Programmers must provide a parameterless function that reads from memory mapped sensor locations and writes to similarly mapped actuator "registers". Code must be "bracketed" by initialization calls that subscribe the controller program to the simulation and cleanup code that is called on simulation exit.. The principle difference in the body of the code is in reading and writing to memory. In C this is often done by mapping a pointer to a memory address and then simply assigning to the pointer or reading from it. These statements have to replaced by the "peek" and "poke" functions of the API. Normally this simply means replacing the assignments in the "real" code with compatibility functions that bind the API names into the code for the real robots. Apart from these few changes the same source code is used in the both the simulated and real robots.

The same compiler front end is used for both real and simulated robots with appropriate switches to target the appropriate architecture. The resulting behaviors of the simulated and real robots are then compared experimentally. No formal attempt is made to verify their similarity. Spread cannot be used to provide formal proofs of correctness it is simply meant to provide a useful tool in the design process.

A final and optional job is to add a simple polygon definition to a textual configuration file that enables the simulation GUI to display the new robot in a distinctive way.

We have already accumulated several hundred controller programs.

5.3 Writing simulations of new hardware

The main task here is to write a program that has three main functions:

The first registers the program with the simulation, retrieves parameters from a text based configuration file including the location of its memory-mapped registers, and retrieves a copy of a list of all the static objects in the world and the current length of the time quantum.

The second is called when the simulator terminates and allows the program to perform any cleaning up such as removing temporary files.

The third implements the model itself and is called at the start of each time quantum. At each call the program must retrieve the new locations of all mobile objects and use these together with the list of static objects to calculate its output values. For sensors this is straightforward; it just has to "poke" them into the relevant locations and send an "end quantum" notification back to the world server.

For actuators, the output value represents the new position that the controlled object would occupy in the absence of any mobile obstacles. This, of course may lead to anomalies if some other actuator process has placed another object in the same position. In the real world the objects would have collided and rebounded in some way. At present the provisional positions are sent to a collision server that resolves the anomaly and returns the resultant position back to the actuator process for use in the next time cycle. This is very inefficient - especially when any sophistication is required of the impact dynamics model. In future versions the collision server will put the collided objects "in touch" with each other so that they can use their own knowledge of the physical properties to calculate their new positions after impact. While not obviating the need for a central collision server this should considerably reduce the load on it.

The task of writing actuator programs is eased considerably by the provision of an extensive API containing functions to perform all the interactions with the simulator described above. In addition Virtual Robots provides additional functions that allow the actuator and sensor process to communicate with the display processes so that, for instance, the state of sensors can be given an graphical representation as described in Section 8.3.

As with controller programs we have already accumulated many sensor models, mostly based on the real hardware available in our laboratory but we hope these will soon be joined by models from other laboratories. At present the only actuator models represent the simple two-motor differential-velocity steering system on our vehicles. During the coming year these should be joined by models of manipulator arms, and other devices for picking up and moving objects.

5.4 Writing new GUIs

SPREAD makes this easy by providing many functions to register with and leave the simulation, to register as a member and to leave "process groups" such as the display group, to send and wait for architecture independent messages, to start, pause, and reset a simulation run, to terminate properly the engine itself and all the processes on participating machines, to read and write configuration files and so on.

6. The SPREAD modules in Detail

6.1 World Server

This is the core of the simulator. Its functions are to:

- Spawn the collision control process
- Set up linkage with display processes During a simulator run the world server can register or de-register display processes. Registered display processes will then receive information about the current simulation state. It is up to the display processes how they present the information to the users.
- Spawn robot processes and moving obstacle processes.
- Control simulation time. The world server synchronizes all process by sending each process in the process list a "start time quantum" message. When it has received a "quantum complete" message from all processes it can then issue the next "start quantum" message.
- Accept requests from console. The world server accepts and acts on simulation control command messages from the console.
- Maintain the global database. This includes the configuration of the robots and the map and the most recent locations of robots and moving obstacles.

6.2 Collision Control Server

This server resolves collisions between moving objects. Since objects have a local copy of the static map they can resolve collisions with static obstacles themselves. At the start of a simulation, the collision control server receives from the world server a copy of the static map and a list of all the moving objects. During each time quantum, the server will wait for positional updates from all moving objects. These indicate the position to which each object intends to move. The server checks for intersections between the bounding polygons of all objects. Where such collisions are detected, the server will use its impact dynamic models to calculate the post-collision positions and send these back to each object and the world server.

6.3 Console

The console process provides simulation control. There can only be one console although, like all modules, it can be located on any participating machine. The principal commands that it can receive and act on are to start, pause, reset or terminate a simulation run, and to change the length of real time represented by each time quantum, thus providing control of the granularity of the entire simulation.

6.4 Displays

Display processes can "subscribe" or "unsubscribe" to the simulation at any time and more than one can be connected at any one time. This allows the simulation state to be presented in different ways at the same time; from different workstations, from different viewpoints, in different graphical forms (e.g. 2-D or 3-D), or as alphanumeric dumps for later analysis and so on. Examples from Virtual Robots will be described below.

6.5 Memory

As mentioned in the previous sections, the purpose of the memory model is to provide communication between the actuators, sensors and controller within each robot. It can model memory mapped I/O such as found commonly in many embedded systems (for example those based as are the Brooker Lab Vehicles on the VME buss. Memory processes simply receive "poke" commands to place a value at a numbered location and "peek" requests to return the value currently stored at a given location.

6.6 Actuators

At the start of the simulation, each actuator receives the location of all static obstacles. In each time quantum, the actuator will "peek" the memory locations that represent the control registers of the real hardware that have normally been placed there by the vehicle's controller process. The model then provides the position it expects the controlled object to occupy at the end of the current time quantum using a forward kinematic model of the combined effect of the characteristics and positions of the simulated vehicle's actuators such as wheels and motors. This model may, if desired, include "noise" effects such as wheel slippage. It then checks if there is a collision with the static obstacles before sending the new location to the collision control server to check for collisions with dynamic obstacles. The collision server will return the post-collision location that it will use in the initial conditions of the next time quantum. The same information is also sent on to the world server so that it can update the global database that will in turn be passed to the display and sensor processes.

6.7 Sensors

Sensor processes provide environmental information to controller processes. The world server provides this information on request to the sensors. This includes the locations of static obstacles (only sent once a start-up), moving obstacles and IAVs. How this information is interpreted depends on the type of sensor. For example, an ultrasonic sensor calculates the distance of the nearest objects in its field of view. Current sensor models developed for Virtual robots include an ultrasonic range finder, an active infrared beacon direction finder and an odometer as well as the track follower described below. To provide compatibility with programs developed for the real vehicles a memory mapped model is used in which the sensors' output values are "poked" to a memory process for later retrieval by controller processes.

6.8 Controller

This module's function is to control the robot behavior by interpreting data from the sensors and issuing commands to the actuators. Controller code is normally shared by both simulated and real robots; much effort has been made in Virtual Robots and SPREAD APIs to allow as few changes as possible when controller code is moved from one to the other.

We also felt it important that no restrictions were imposed on the size or complexity of controllers. In practice controllers have ranged from simple "reflex" behaviors like "turn left on impact", through hierarchical fuzzy rule-based systems [6] to current work on PROLOG "plan-while-executing" planners.

6.9 Moving Obstacles

The moving obstacle processes are no different in principle to robot process sets except that they are usually implemented more simply as single processes. Currently Virtual Robots only includes simple objects that move through the environment at a constant velocity.

6.10 Active Targets

Active targets model beacons that emit detectable signals. In the Brooker Laboratory we have implemented some simple active beacons that emit identifiable infra-red signals. The bearing and ID of these (but not the distance) can be detected by the robots.

7. The Virtual Robots Interface

SPREAD itself only provides a machine independent computational engine. To be useful as a teaching tool an interactive graphical user interface must be provided. The Essex University Brooker Laboratory is equipped with PCs that run the NeXTSTEP operating system and so the Virtual Robots front end to SPREAD that we use at Essex as inevitably been developed using that system. A fuller description can be found in [7]

The device-dependent part of Virtual Robots comprises an interactive, graphical configuration file editor, a simple console panel and a graphical simulation view window.

7.1 The configuration file editor

The purpose of this is to set up the initial conditions for a simulation run. It uses the "drag and drop" paradigm to place and parameterize robots and active targets. When complete it will also allow the placing of all object types such as obstacles and followable tracks which at present have to be entered into the configuration files by hand. We also intend to take a similar approach to the "building" of virtual robots so that a kit of robots parts can be assembled interactively.

Figure 7-1 shows the editor in action. The smaller polygon represents an obstacle and the larger one represents the a track that can be detected and followed. The icon at bottom left has been used to add several robots by dragging and dropping in the desired position. When this is done the user is asked to supply the path to that robot's configuration file. Once placed the arrow representing the robot's initial heading can be dragged to point in the desired direction.

Active targets can be placed similarly by using the "lighthouse" icon. Double clicking on the placed target brings up the "Goal Parameters" panel shown in the illustration. This allows the identification code, aperture angle and range to be selected. These are also shown in the view window although this is not apparent in the illustration. Clicking on any object also causes its current settings to be shown at the bottom of the window. Also not shown is a menu to load and save configuration files.

Robots and targets can be removed simply by dragging them out of the editor window.

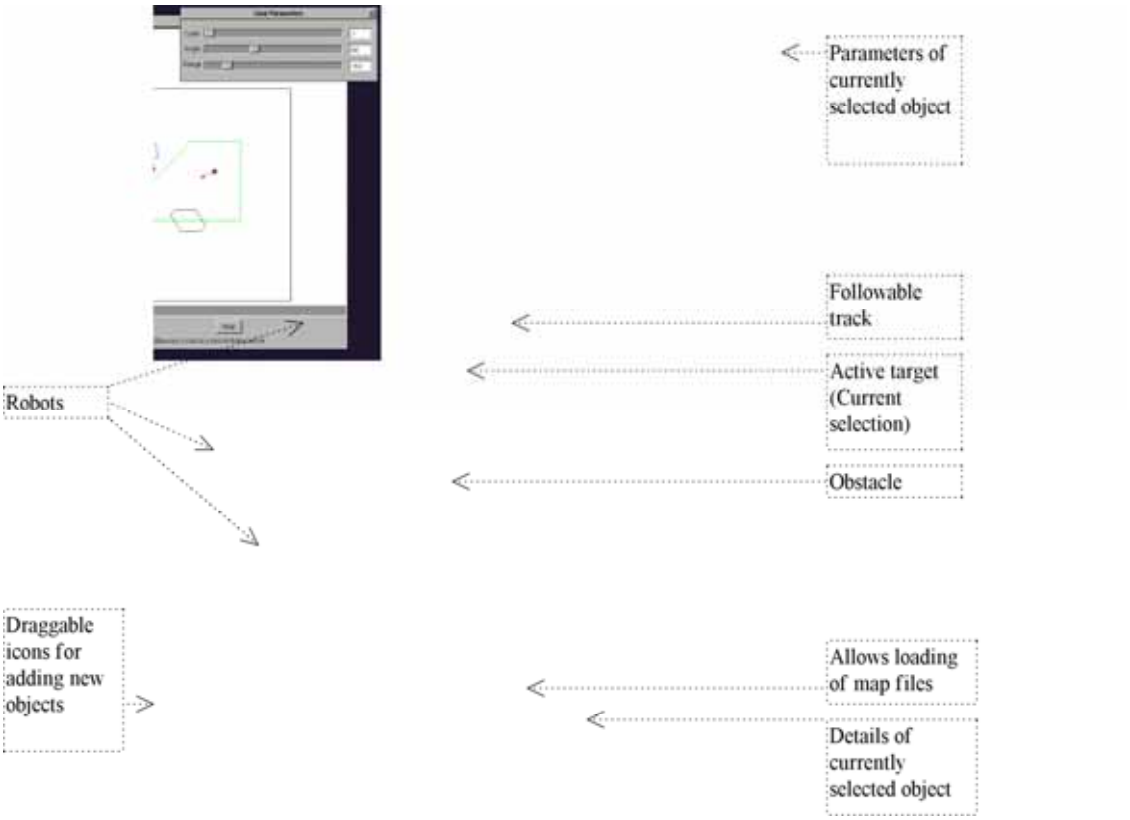

Figure 7-1 A screen dump showing the interactive configuration file editor

7.2 The Virtual Robots Simulation Viewer

This is the most developed part of Virtual Robots and has many facilities to help users to see the world as the robots see it by making the sensors' fields of view and their outputs visible. Some of these facilities are shown in Figure 7-2 below. The depicted robot has started at the top right of the picture moved in a straight line until detecting and following the track. Near the bottom left it has detected the obstacle and switched from track following behavior to an edge following behavior. The test run subsequently shows an error in the control algorithm where rather than reverting to track following as intended when the track is found again it continues to follow the edges of the obstacle.

Two methods of tracing the robots' routes are given: a trail of dots or, as in Figure 7-2 a trace of the robot's bounding polygon. The particular robot shown was equipped with eight ultrasonic range finders. These only give the distance to the closest sensed object within the sensing angle. This is shown by drawing both the circle segment representing the area that can be sensed by each sensor and the arc representing the possible positions of a sensed object. This can most clearly be seen as the robot in the picture senses the obstacle on its left hand side. An alternative method of showing the possibilities of accumulating the range data is provided where a trace is left of the arcs where an object may possibly be. This has proved invaluable in illustrating methods of building maps from such cumulative and uncertain data such as the popular "occupancy grid" methodology pioneered by Elfes [8] Other sensors' activity can be viewed by similar methods.

Options are also provide for zooming into particular areas for more detailed observation and for scrolling around the simulation area (see Figure 8-1 below)

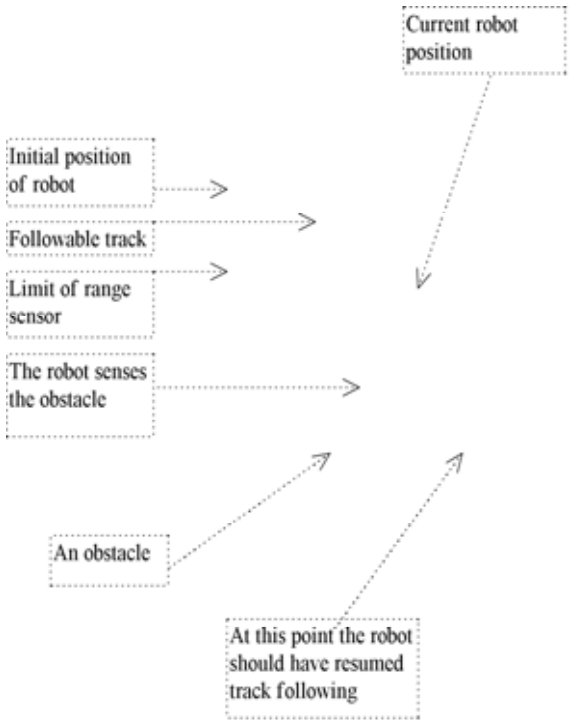

Figure 7-2 A screen dump of a simulator run.

8. Case Study

In order to illustrate how the simulator may be used in practice we will now describe in some detail the work of a third year undergraduate who used the simulator to develop a new sensor for our IAVs. The student followed the classic design cycle in that he first developed a mathematical model for the sensor and then implemented this in the simulator before any construction was attempted. The student was able to experiment with the simulation by programming an algorithm that used the sensor. This could be tested, still under simulation, in order to determine the most effective dimensions of the final sensor. The actual sensor was then built, and in this case, performed almost exactly as predicted. The sensor model is now available as a component both in reality and in the simulation with some confidence that the behavior will match in both cases. For example experiments on the simulator can be used to predict the emergent behavior of a large number of IAVs equipped with the new sensor.

The sensor in question comprised an array of photo-reflective emitter/detector pairs that was to placed under the vehicle so that it could detect areas of high reflectivity on the ground. In particular the sensor was to enable the following of narrow tracks of reflective material placed on the floor. Part of the brief was that the sensor was to be able to detect corners, junctions and end points in the track. A typical test mission for the IAV was to search for the track and then exhaustively search the track network..

At present the simulator represents the world in two dimensions (although the promotion to three dimensions is the subject of present work). Nevertheless the student was able to structure the sensor model in 3-dimensions and take into account the vital parameter of the sensors' height above the ground and the possibility that the each sensor may not point directly downwards. We always encourage students to avoid unnecessary specialization of their models to allow for possible future unforeseen uses for their hardware.

8.1 Modeling the sensor

In this case the student chose to model each IR transceiver as a separate process. On the assumption that the part of the world visible to the sensor formed a cone, the model involved the calculation of the ellipse formed by the intersection this cone and the ground (Figure 8-1). Having obtained this ellipse the problem could then be treated as the two-dimensional one of calculating the intersections of the reflective track objects with this ellipse.

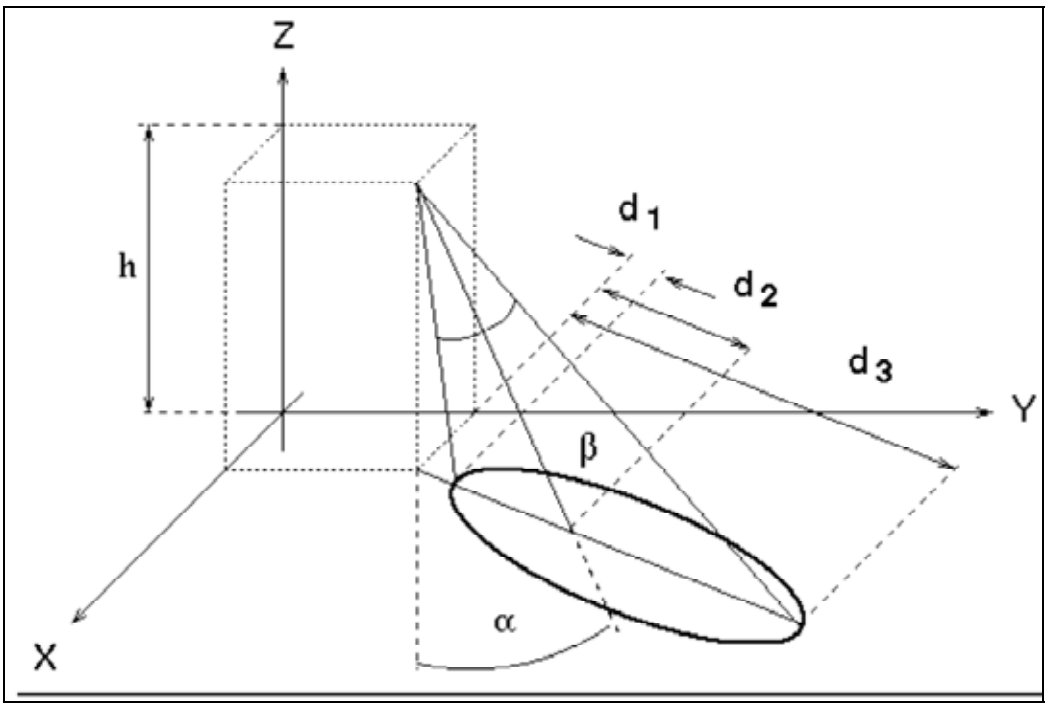

Figure 8-1 The sensor parameters are the height h, the inclination α and the cone opening angle β.

Further sophistication was added by not treating the entire area of the ellipse uniformly to allow for the non uniform amount of reflected light when the sensor is not perpendicular to the floor surface. In these cases reflective objects entering the more distant parts of the ellipse are less likely to be detected. The student's solution was to divide the ellipse into a number of segments of different sizes with segments more dense in areas of greatest sensitivity (Figure 8-2). Intersections are calculated for each segment and only if a simple count of all the covered segments is greater than some threshold does the model output a 1 to represent the high bit output by the real sensor on detection of an area of high reflectivity.

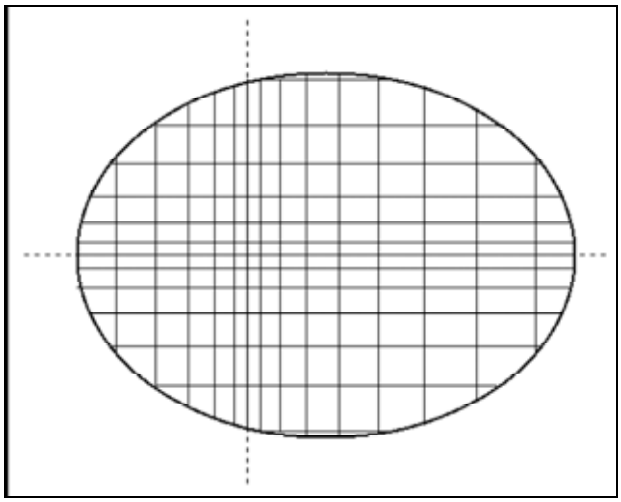

Figure 8-2 The distrubution of illumination. Each rectangle receives an equal amount of light.

8.2 The sensor configuration file

In order to integrate the model into the simulator two things have to be provided. The first is a program that implements the model that has to be written in C or C++ and secondly a text description file that provides the parameters for a particular incarnation of he model. A typical configuration description for the sensor looks like:

```
Sensor = {

          Height      = 8

          Inclination = 45

          Cone Angle   = 30

          Precision   = 16

          Coverage    = 15

          Max Value    = 255

          Error Level  = 5

          }
```

`Height`, `Inclination` and `Cone Angle` are the geometrical parameters previously described (**h**, α and β respectively). `Precision` represents the number of segments to be created within the elliptical sensing area.

`Coverage` (expressed as a percentage) takes into consideration the intrinsic infrared sensor physics. A real device triggers only if a minimum percentage of the perception field is illuminated by reflected rays. As a result a y reflective body under the sensor is detected only if its dimension is bigger than this threshold or if it is able to reflect back to the infrared receiver a sufficient amount of radiation.

`Max Value` is the maximum possible reading. Every computed value is normalized on it.

`Error Level` (expressed as a percentage) represents noise within the sensor emulation. There are a number of noise sources that can affect the physical sensor such as fluctuations in the reflection/absorption coefficients, cross-device interference, power-supply and other intrinsic noise effects. This parameter adds the defined level of random noise.

8.3 Virtual Robots Interface

The final task for the student was to provide a visual representation for the display. Each sensor is shown as two small concentric circles when the track is detected in their field of view. This can just be seen in the screen dump (Figure 9-3) which depicts two robots attempting to find and follow the track. Robot number 1 has just found the track and is turning to follow it. Robot number two has been following the track for a short distance but has already detected robot number 1 with its ultrasonic range sensors and is turning left to avoid it.

9. The results of the case study

Having decided on a suitable sensor element the student was able to use the simulator to design the a suitable physical configuration for the entire sensor, as well as its placing on the robot. He could also experiment, in the virtual world, with different widths of track material in order to determine the narrowest track that would be reliable. He was even able to start developing algorithms for controllers that could plan routes from one point to another along complex tracks and to guide the robots along them.

We were very gratified to find that once a completed sensor had been constructed and mounted on a robot its behavior matched that of a single simulated counterpart very closely - including failures where the track was narrower than the designed-for dimensions.

We have yet see whether multiple real robots would behave as in Figure 9-3 as at the time of writing we have only built one prototype track following sensor.

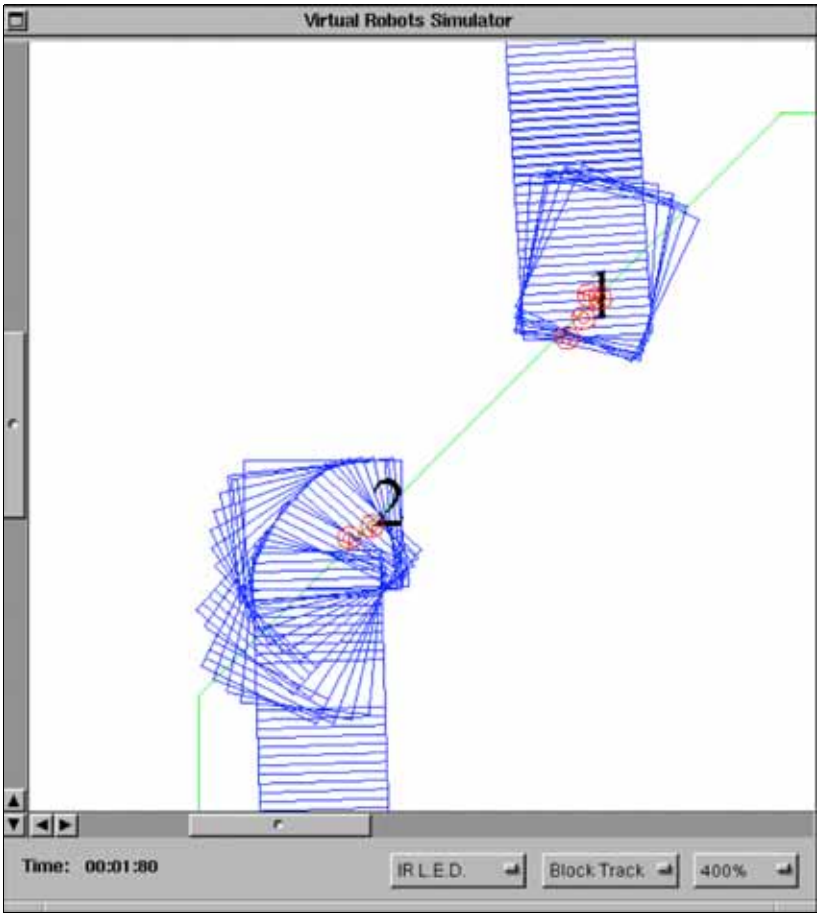

Figure 9-3 Two robots attempting to follow the track and about to take avoidance action

The prototype sensor is shown in Figure 9-4

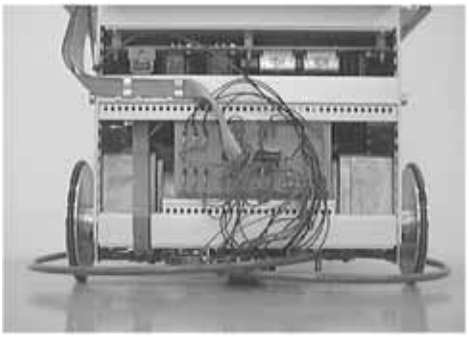

Figure 9-4 The new sensor mounted on a robot

10. Performance Analysis

Throughout the development of the SPREAD project we have striven for simplicity, portability, transparency with respect to programming, and ease of use. In particular this has led us to make the simplest of mapping of "objects" to processes. Ideally, to preserve transparency this should be without regard to the placing of processes on machines and the

computational and communication demands of each process. Finding practical ways to overcome the performance inefficiencies that arise from this naïve approach is to play an important part in future development of the project. As a first step we have made some tests to establish a benchmark against which to measure future improvements.

The testing that is described below was performed on a typical cluster of 12 '486 DX2 66MHz PC's running NeXTSTEP. The network was a standard Ethernet isolated from the rest of the campus network by a managed bridge. The experiments were run at night when none of the machines were being used. The tests were only of the SPREAD engine and none of the graphical front end was involved. This has also allowed us to perform similar experiments on other and mixed architectures. Unfortunately this data was not available at the time of publication of this article although first indications are that there was very little difference between the PCs, SPARC 5 Sun workstations and even DEC alpha 200MHz machines. This seems to confirm ones' intuitions that the naïve placing of processes on random machines renders the system I/O bound and the 10MB/s Ethernet is the system bottleneck.

All tests were run at least 10 times and the quantities in Figure 10-1 represent straightforward means of all runs. The method was to run a typical simulation with on a varying number of machines and simulating a varying number of robots. All robots were configured identically and were running the same control program. In each case the simulation was run for a fixed number of time quanta representing the same amount of simulated time. Performance was measured by the mean (real) time taken per time quantum.

Table 10-1 shows the number of processes spawned for each test.

Robots	Processes
1	13
5	41
10	76
15	111
20	146
25	181
30	216
35	251
40	286

Table 10-1 Total number of processes spawned for each test.

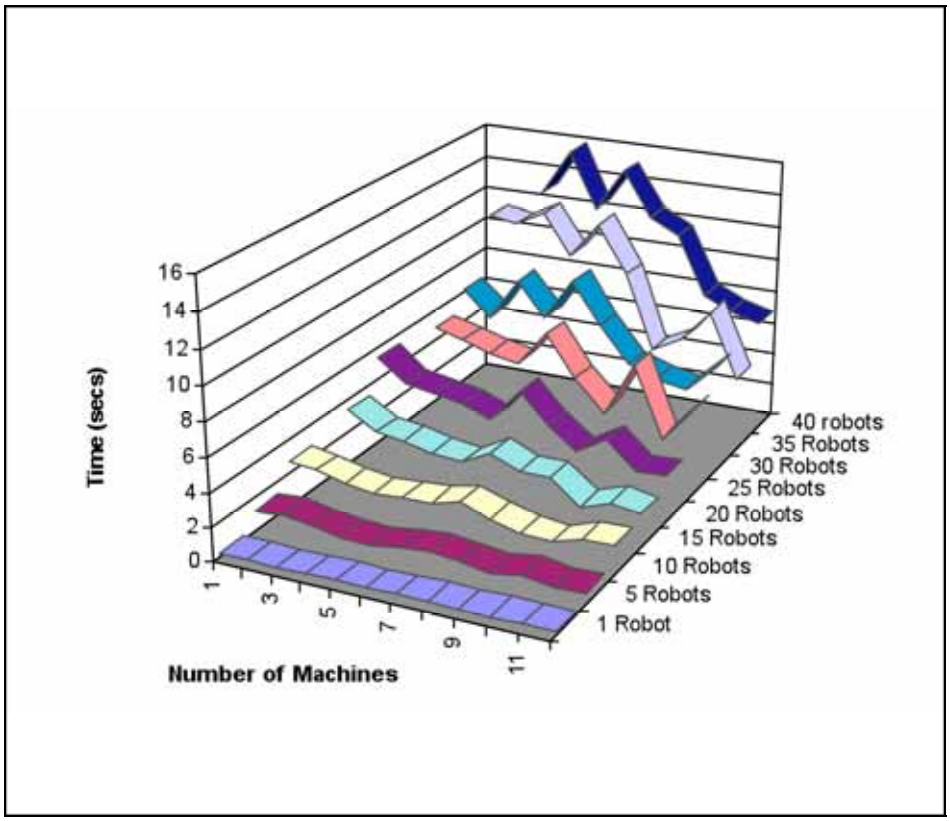

Figure 10-1 Average time per quantum using different numbers of computers and simulated robots

The general trend of the curves met our expectations; times obtained for more machines are generally shorter than for fewer machines. However there are many cases were adding machines actually make the performance worse. It can be seen from Figure 10-1 that the best performance is for six machines whereas the worst is for seven. The traffic between controller and memory is the key to this anomaly. In the current implementation no attempt is made to place processes in the best places so it is quite possible that processes that communicate a great deal, might be placed on different machines. In the 6 host case controllers and their associated memory processes happened to be placed on the same physical nodes and the network traffic did not play a prominent role. With one more machine (7 hosts) memories and controllers tended to be placed on differing machines needed the network to communicate, creating an I/O bounded system. The best effective speedup we obtained was for 35 robot simulations which ran about 5 times faster on 12 machines than on one.

We conclude from this that we need to:

- place processes more intelligently - maybe simply placing all processes for a robot together would be sufficient;
- reduce network traffic - use of true multicast will be a great improvement as much information such as updates to the world database could be read simultaneously by many processes;

- reduce network collisions. Switching hubs are now inexpensive. Even at a nominal 10Mb/s with one machine on each switched outlet an order of magnitude increase in total bandwidth. This would also enable gains to be made by delegating collision processing to the colliding actuator processes. At present the collision server is a bottleneck.
- use faster network technology. We will soon be installing fast (100Mb/s) Ethernet in the laboratory with uplinks to FDDI that will eventually link to other laboratories making available very large numbers of machines (several hundred) all at high bandwidth. Plans are afoot (as they are no doubt in most institutions) to move to ATM technology that should be ideal with its high speed and low, deterministic latency.

11. Future work

The greatest performance gains are likely to come from the more intelligent placing of processes on processors. Providing a means for users to manually place processes will be relatively easy to implement and we hope to develop more sophisticated monitoring of performance statistics in the hope that they could be used to automatically generate optimal process placings for future runs. The Holy Grail would be an adaptive system that could use performance statistics and the ability to dynamically migrate processes and "learn" better placings on the fly.

Although PVM gives high portability and a simple programming model it has many inefficiencies in its implementation and search for alternatives are a high priority.

Work is underway to make the underlying world model fully 3-dimensional. Virtual Reality type viewers are being constructed to take advantage of this.

We are investigating faster network technologies such as 100 MB/s fast ethernet and switched concentrators that should make the network bandwidth an order of magnitude more favorable for very low cost. It will be interesting to see how the performance curves are affected although the intelligent placing of processes is likely to remain critical for the foreseeable future.

12. Summary

The high quality of work completed by many students over the two years that the simulator has been in use have convinced us that there is much to be gained by making this type of simulator more widely available to computer science students. In particular, where robotic hardware is available simulation increases the number of students that can benefit from it as well as improving the quality of the limited access that they do have.

The advantages to be gained by using cluster computing methodologies to improve performance remain tantalising but still largely unrealised. The automatic placing of processes on processors to achieve optimal performance remains elusive. Meanwhile, recent developments in low cost, fast networking such as 100baseXX fast Ethernet, gigabit rate switching technology and ATM promise to make some modest inroads into redressing the imbalance between workstation computational speed and the speed of communicating between them.

For the latest information have a browse around our entry in the European Robotics Archive at http://cswww.essex.ac.uk/Eurobots/simulators.html.

13. Bibliography

[1] Parallel Virtual Machine (PVM), Oak Ridge National Laboratory, Oak Ridge, Tennessee. The home web page for PVM is at http://www.epm.ornl.gov/pvm/

[2] P. van der Smagt (1994) `` Simderella: a robot simulator for neuro-controller design," Neurocomputing 6 (2), Elsevier Science Publishers, pp. 281-285

[3] Webber A.D. Xmouse. Details at ftp://ftp.essex.ac.uk/pub/robots/Simulators/Xmouse

[4] Douglas C. MacKenzie, Jonathan M. Cameron, and Ronald C. Arkin, "Specification and Execution of Multiagent Missions," July, 1995. Available at http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab

[5] O. Michel, The Khepera Simulator, University of Nice, France. Details can be found at URL http://wwwi3s.unice.fr/~om/khep-sim.html

[6] Voudouris C., Chernett P., Wang C., Callaghan V. "Hierarchical Behavioural Control for Autonomous Vehicles", 2nd IFAC Conference on Intelligent Autonomous Vehicles, Helsinki University of Technology, Espoo, Finland, June 12-14, 1995

[7] Ching, Calvin K.W, "The Virtual Robots Simulator", Dissertation (MSc) Department of Computer Science, University of Essex, England, 1994. Copies can be requested by email to robots@essex.ac.uk

[8] Elfes, Alberto, "Occupancy grids : a probabilistic framework for robot perception and navigation", 1989. Thesis (Ph.D.), Carnegie Mellon University. Photocopy. Ann Arbor, Mich. : UMI Dissertation Information Service, 1994.