Simulation of small autonomous mobile robots

Axel Waggershauser

Dec 2002

Author:	Axel Waggershauser
Supervisor:	Prof. Dr. Thomas Bräunl, Prof. Dr. Ewald v. Puttkamer
Organisation:	Universität Kaiserslautern
	Fachbereich Informatik
	AG Robotik und Prozeßrechentechnik
	Postfach 3049
	67653 Kaiserslautern

Contents

Chapter 1

Introduction

The EyeBot controller is developed to be a very versatile module, suitable to build lots of different applications in the context of autonomous mobile robots, like small 2-wheel robots, omni directional robots, more leg robots or humanoids. An overview of the capabilities and the different applications of the EyeBot controller is presented in [1] and [2]. One popular application is the soccer playing robots. A typical soccer robot is a small (about 10 cm diameter) 2-wheel robot, equipped with different sensors like distance giving infrared sensors (PSD), binary infrared obstacle detection sensors, possibly bumpers, a small color CMOS camera. Besides the sensors, it is equipped with a set of servos as actors, which are used to pan and tilt the camera and to implement a kicker mechanism to kick the ball. Figure 1.1 shows an example of the current hardware.

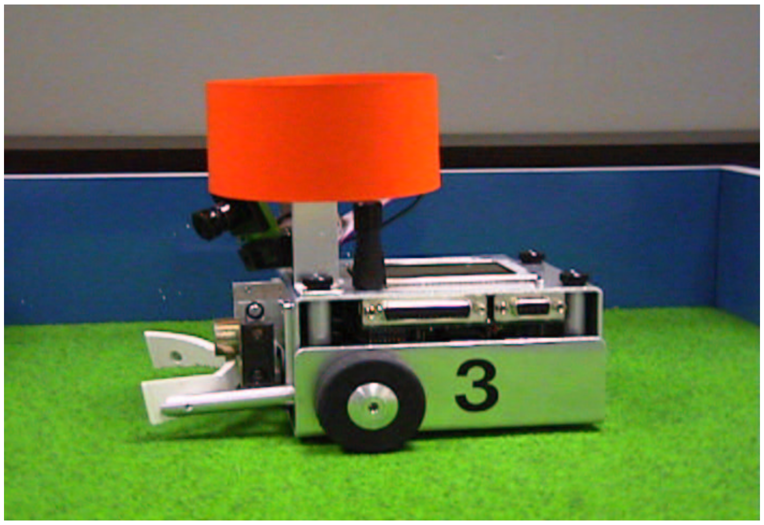


FIGURE 1.1: A real-world soccer robot

The simulation environment, which this thesis is about, deals with exactly this application. This type of (soccer) robot will be called “robi” in the following. Please note that for convenience sometimes “robi” is also used for the program executed on the robot. The aim of this research was to develop a system, that allows one to execute a program, written for a robi, as is, in a simulation environment, capable of simulating all different sensor inputs including the camera. As a simulation without being able to monitor its progress is quite useless, the system has of course to enable the user to monitor the simulation including the movements of the robis and the content of their LCD display. An important feature of particular interest is to be able to simulate multiple robots simultaneously in one simulation, interacting with each other, which is obviously essential for the soccer application. “As is” means, that the source code of the program can be used without any changes or restrictions. So this simulator can be used, as a developing and testing environment for higher level control programs and provides all advantages of a usual simulation environment:

- slow motion and time laps effects
- the ability of debugging with a usual debugger on the host machine
- simulating an arbitrary number of robis (limited only by the performance of the host system)
- creating arbitrary environments, in which the robis act
- simulating with the absence of (or arbitrary) sensor errors

The application programming interface, to control the EyeBot is the RoBIOS. The word RoBIOS is an abbreviation for Robot BIOS or Robot basic input/output system. It is merely a set of (about 100) C functions, one can think of it as a library. A program for the robi is a usual C program, that can use standard C library functions and of course RoBIOS functions. To achieve the goal of being able to simulate (execute) ordinary robi programs, it is necessary to provide an implementation for the RoBIOS functions against which the program can be linked. These functions are the only connection between the program to simulate and the simulator. The only way for the program to influence the environment is through the RoBIOS functions and the only place for the simulator to influence the execution of the program is the implementation of these functions.

In this thesis the underlying models and different aspects of the implementation of the simulator are described in detail. First the software structure is described, introducing the logical parts of the software and their inter relationships. Then the time model, used in the simulator, is described. The emphasis of this thesis lies in particular on this time model and the related models for synchronization and multitasking, as these are identified to be the crucial parts for achieving the goal of simulating multiple robis simultaneously. Then the models describing the world or environment and the models describing the different aspects of the robis are described. Following is a discussion of the simulation of the different functional parts (the sensor and actor simulations). At the end a brief manual of the simulation software and some implementation details are given.

Chapter 2

The model of the simulation

2.1 The structure of the simulator

The global structure of the simulator is client-server based with an object oriented design. The central part of the server is one object of the class called Core. The clients are objects of the class Robi. There is one Robi object for each simulated robi. I will simply call these objects "robi" as well, since there is a one-to-one correspondence. The robis communicate with the core via a message passing interface, i.e. they exchange messages via a bidirectional message channel.

The core has the following responsibilities:

- initializing the simulation and starting the robi clients
- maintaining the state of the world, including the current position of all robis and the balls
- synchronizing the robi programs
- distributing inter-robi messages like the radio messages
- maintaining the speed of the simulation (time lapse and slow motion effects)
- starting, stopping and pausing the simulation triggered from the user monitor (GUI)

The clients are constructed with the aim in mind to minimize the necessary communication with the core for better performance and to realize a clear modularization and information hiding. So all RoBIOS functions that can be simulated with locally, in the client available information, like functions to retrieve the current time or the current speed, are implemented without communicating with the core. Only if the client takes an action that possibly influences the other clients, a message to the core has to be

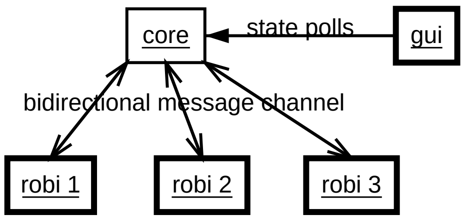


FIGURE 2.1: The client server based structure of the simulator

sent. This includes changes of the robi's velocity and sending radio messages to other robis. And of course for synchronization, there has to be sent a message back and forth.

With each robi is associated a program, that should be simulated. Therefore each robi client has to be an active component, capable of executing the robi program. This could be a thread for each robi in one process or a single process for each robi (or even a mixture of both).

The reason why I chose a client-server model with a message passing interface is because it allows to easily separate the implementation of the core and the robis in different processes with no shared memory. These processes can be executed on different host machines communicating over a network, so it is possible, for example to have two teams of robis playing soccer with each team on a different computer.

The current implementation is based on threads instead of processes, since threads have the advantage of easier handling during implementation and using and the performance is probably better and definitely not worse. As a matter of fact, it allowed me to take advantage of the shared memory as well. Thus, there would be some work necessary, to remove the direct access of the robi clients to the world state information, if one wanted to move from the thread implementation to a process based implementation. Some more details about the actual implementation can be found in chapter A.

The core is designed to be completely independent from the user monitor. The simulation can principally run without the user monitor. The idea is that the user interface can poll information from the core asynchronously, this prevents a slow display method, e.g. with a complex scene and no hardware accelerated graphics adapter, from unnecessarily slowing down the simulation progress.

2.2 The time model

This section discusses the way time is modeled in the simulation system. There are two reasons why an appropriate modeling of time is a prerequisite for building the described simulator. First of all, the simulation includes a dynamical model for the movement of the robis, i.e. simply, that the robis are moved by the program by specifying a linear and an angular velocity, according to which the robi has to be moved in the simulation over time. To properly simulate one robi it is necessary to be able to determine the current time, and hence, the current position, hence the sensor values, whenever they are requested. A more detailed description of the used dynamic model will be given later. Second of all, the robis are able to influence each other in basically two ways, on one hand, they can influence each others simulated sensor values by moving in the simulated world and on the other hand, they can use radio communication to directly exchange messages between each other. And here it gets clear, that not only a proper model of time for each single robi is necessary, but also a model for synchronization of the single robi programs. It has to be made sure, that every time two robis interact with each other, they have to be at the same point in (simulation) time. Otherwise messages from the future and similar inconsistencies would be possible.

I distinguish between different concepts of time in this context, they are listed next.

- *virtual time (of a robi)*:
  Each robi has its own *virtual time* associated to it, it is the "life time" or the current time of the robi, the amount of simulated time, e.g. if the *virtual time* of a robi is 10 seconds, then in the real world the start of the program would have been 10 seconds ago.

- *simulation/global time*:
  This is the current time of the whole simulation, it is analogous to the *virtual time* associated to the robis. Actually it is just defined as the *virtual time* of that robi with the smallest *virtual time*.

- *real time:*
  This is the one, easiest to understand, it is simply the time that was used to execute the simulation on the host computer, i.e. the time that passed in the real world since the start of the simulation. If one wants to have the simulation running as fast as the real robots would execute the programs, then the *real time* and the *simulation time* would have to be kept equal during the simulation.

2.2.1 The *virtual time*

For a correct simulation it is necessary to keep the *virtual time* of all robis sufficiently synchronous. The *virtual time* of a robi is always then of importance, when the robi program calls a RoBIOS function, e.g. to perform a sensor measurement. The following paragraphs will describe how the *virtual time* of a robi is maintained.

The *virtual time* of a robi at a certain state of the control program execution is basically the time, that the robi's CPU needed to execute the program up to this point, i.e. *virtual time* increases during execution of machine code and possibly during waiting for external events/interrupts while measuring a sensor, for example. It is not necessary for the simulation to deal with this low level. In the simulation there are only two different contributors to the *virtual time*:

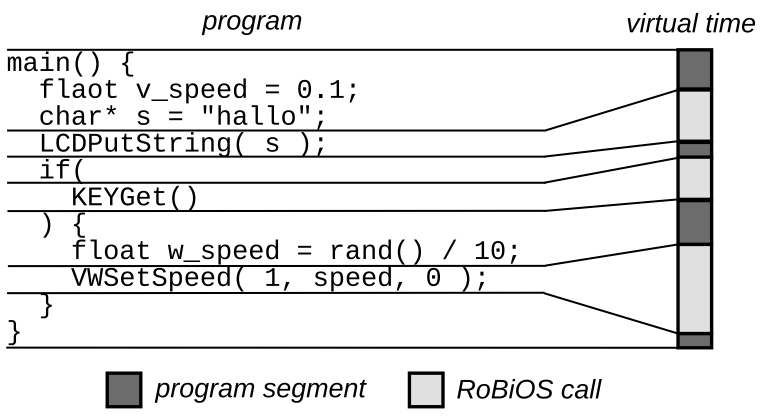
- the execution of a RoBIOS function,
- the execution of "normal" code, i.e. outside of a RoBIOS function.

The amount of time, that each RoBIOS function call needs on the real robot, can be measured on the real robot and stored in a table for simple lookup during the execution of the simulation. Every time the program calls a RoBIOS function, there is a table lookup to determine the time, that this call consumes and this time then gets added to the current *virtual time* of this robi, which is simply stored in a variable for each robi.

The other contributor is more difficult to find out, since there is no direct way to get the amount of time the robi CPU would need to execute a piece of code, that the simulator does not even know (he can just execute it).

To clarify what is meant by this "piece of code" and for easier referring to this in the future, I want to introduce another concept, the "program segment". As already mentioned, a control program for the robi is composed of a sequence of RoBIOS function calls mixed with or embedded in "usual", computational code. From the point of view of the simulation environment, the possibly complex structure, in terms of nested loops, etc., of the program, can be seen as a simple sequence of alternately a RoBIOS function call and what I call a "program segment". One program segment is the possibly empty set of machine code instructions, that has to be executed between two consecutive RoBIOS function calls. The program segment is empty if there is no other C-command between two successive RoBIOS function calls. This is illustrated in figure 2.2.

An approximation for the time, that the robi CPU needs to execute such a program segment, can be made by measuring the time, that the host CPU needs to execute this program segment. This is done by starting a timer on entry of a program segment, which is on exit of a RoBIOS function and stopping on exit of this segment, which is on entry of the next RoBIOS function. Then this time is passed to account with an appropriate factor, that describes the ratio between the robi CPU speed and the host CPU speed. It is only an approximation, because this ratio is not necessarily definite, since the host might have a FPU but the robi has none and because the usage of the host CPU time can only be gathered quite imprecise (for this application). A typical resolution is a hundredth of a second (Linux kernel 2.4.10, use of the function times() or getrusage()). With a ratio of 50 (25 MHz to 1.25 GHz), we only have a robi CPU time resolution of half a second, which is quite useless. I believe I actually have to think about another way to gather this *virtual time* contributor.



```
main() {
  flaot v_speed = 0.1;
  char* s = "hallo";
  LCDPutString( s );
  if(
    KEYGet()
  ) {
    float w_speed = rand() / 10;
    VWSetSpeed( 1, speed, 0 );
  }
}
```

FIGURE 2.2: The *program segment*

2.2.2 The global simulation time

The global time, in contrary to the *virtual time* of each robi, increases in discrete time steps of T seconds, in the current implementation T is set to 0.05 seconds. Thus the state of the world, including the information where the robis are, gets updated every T seconds of simulated time. The *virtual time* of a robi can be assigned a time slot $ts(\textit{virtual time})$, which is the biggest integer I, such that $I * T \leq \textit{virtual time}$.

This discretization of the global simulated time has some potential consequences for the simulation results that I want to point out. The state of the world changes only every T seconds simulated time, so if a robi program would, e.g. request two images from the camera interface within a time interval of less than T seconds, the two images would be identical, since the positions of the robis would not have been updated between the two measurements. But since the capture process for one image takes about 0.3 seconds on the real robot and therefore in the simulation as well, this is no problem. After the image capture RoBIOS call returns, the *virtual time* of the robi has increased 0.3 seconds, so two successive calls of the capture function are at least 0.3 seconds apart. The same applies to other measuring processes depending on the state of the world. The distance giving PSD measurement, for example, has an update rate of about 0.06 seconds.

2.2.3 Synchronization of the robi programs

As mentioned already, the only link between the simulated robi program and the simulator is the set of RoBIOS function calls. So the simulator can only suspend the robi program execution when it enters a RoBIOS function. I.e. the execution of a program segment can not be interrupted, even if the execution of this program segment takes a lot of *virtual time*. But this is no problem, since the execution of the segment is not dependent on any external data. In other words, the program segments are "atomic" but they can be executed "in advance", no matter how long their execution takes.

The synchronization of the robi programs is done by selectively stopping the execution of the programs until all of them reached or over stepped a certain *virtual time*, which is the current simulation time.

The realization is based on a bidirectional message exchange between the different robi threads and the core of the simulator. Each robi has besides his *virtual time* another time associated, the time of its last synchronization. On each RoBIOS function entry, the current *virtual time* is determined and compared with the last synchronization time. If the *virtual time* lies not in the same global simulation time slot than the last synchronization time, i.e. $ts(last\ synchronization\ time) < ts(virtual\ time)$, then this robi thread has to be synchronized with the simulation core. It sends a synchronization message with the current *virtual time* as parameter to the core and waits for a wakeup message before it proceeds.

The core maintains a list containing the *virtual time* of each robi. Each time the core receives a synchronization message from one of the robi clients, it updates the *virtual time* for that robi in its list and then finds the minimum *virtual time* in the list (mvt). If $ts(mvt)$ is bigger than the $ts(current\ global\ time)$, i.e. all robis passed the time $global\ simulation\ time + T$, then the simulation proceeds one step: the global simulation time is set to $ts(mvt) * T$, the state of the world gets updated, which means all robi positions have to be updated for the new global simulation time, including the collision tests. Then the core sends wakeup messages to all robis, whose *virtual time* is in the same time slot than the new global simulation time. Thus, when the robi client resumes, its robi's *virtual time* is in the time slot of the global time and the robi program is synchronized.

The time lapse and slow motion effects for the user monitor are realized by synchronizing the scaled simulation time with the real time, by simply suspending the core for an appropriate amount of time every time before the wakeup messages are sent. To guarantee a smooth update of the user monitor, the above described algorithm has to be slightly modified. If the step from the old to the new global simulation time would be to big, there are intermediate steps inserted, the world updated and synchronized with the real-time. To illustrate the problem an example is given. If the user sets the simulation speed to real-time and there is only one robi in the simulation and this robi's program executes a turn command, that takes 2 seconds, then, on the next synchronization of the robi thread, the core would increase the global simulation time at about 2 seconds, then sleep for about 2 seconds and then update the display when the robi already finished the turn command. So on the monitor, nothing happens for about 2 seconds, then suddenly the robi has turned.

2.3 The multitasking simulation

After having described the process of synchronizing the *virtual time* of several robis, this chapter is about the synchronization of several tasks performed on a single robi. First a short description of the multitasking capabilities of the RoBIOS is given, including a description of the problems occuring with the simulation of multiple tasks per robi. The two following sub chapters give a detailed discussion of two different approaches to a solution of these problems.

The RoBIOS supports multitasking programs, both preemptive and cooperative multitasking. The usual steps to establish a running multitasking system with the RoBIOS are: first the initialization of the multitasking system with the specification of the mode to use, then the setup of all threads that should be executed. Each thread has to be added to the thread queue, providing the thread function address, the priority, a name, the stack-size and a user id. The thread state has to be set to READY, this is necessary, since the threads are initially suspended. The next step is to initiate the actual task switching. In case of cooperative multitasking, this is done with the reschedule command and in case of the preemptive multitasking, this is done with the permit command, which permits the preemptive task-switching and at last there is usually a subsequent wait call, to keep the CPU busy until a timer-irq occurs, which in turn will perform the task-switch. The main thread, i.e. the context of the main() function, plays a special role. It is not a usual thread like the spawned ones. It does not get rescheduled until all other threads are done or suspended. Normal threads can be killed by themself or by other threads, they can be suspended and resumed by any thread, they can temporarily suspend themself (sleep), they can use semaphores for secure inter thread communication. Another feature, related to the multithreading, is the presence of timers. A timer in this context is, like a thread, a separate function. Unlike a thread, it has to be executed periodically after a specified time interval. Since it is in fact a hardware timer irq handler it can not be interrupted, it simply occupies the CPU until it is done. The simulation environment has to be able to achieve all this functionality.

The biggest problem is the correct modeling and handling of the *virtual time* of the robi. The question is, if we have multiple threads running on one robi, especially if we operate in preemptive multitasking mode, what is the appropriate *virtual time* when a RoBIOS function is entered by one of the threads. The problem will get more obvious after thinking a bit about the possibilities of modeling the robi threads in the simulation. It is suggesting to establish, for each thread to be simulated, a thread on our host machine. To prevent further confusion, I will clearly distinguish between threads in the context of the robi on one hand, i.e. threads, which are executed in a robi and shall be simulated in the simulation. I will call these threads the "robi threads". And on the other hand threads in the context of the simulation, i.e. the entities, that model the robi threads and that are to be executed on the host system. These threads will be called simulation threads, host threads or simply threads. It is therefore suggesting to use threads on the host to simulate robi threads, because they are able to perform context switches, but that is actually pretty much all we can expect from them. At first sight, there is not much to be said against using one host thread for each robi thread, letting them independently execute the appropriate code, maintaining one *virtual time* for each robi and when it comes to the point, where a thread has to determine/process the *virtual time*, it simply accesses its associates robi's *virtual time*, using mutual exclusion with the other threads regarding the access. But since we can not rely on a fair dispatching by the system's scheduler, we can not simply proceed like this without proper synchronization to guarantee that 1) the threads get robi (CPU) time according to their priority and even more important 2) that the time slices are reasonable. It is not acceptable that one thread gets a *virtual time* slice of more than a few hundredth of a second. Exactly for this reason, it is necessary, to think about a model of synchronization, that the simulation of the multitasking feature has to based on.

The idea here is the same as with the synchronization of the *virtual time* between the robis. We can always execute a complete program segment without having to worry about synchronization. The definition of a program segment of a thread is just the same as before. We have to think about synchronization at the moment we enter a RoBIOS function. At the date of the RoBIOS function entrance, we would like to determine the just valid *virtual time* (of this thread) to be able to interact with the environment appropriately. The problem is that we are not necessarily able to do this at this date, since the time passed by during the execution of the program segment depends on the other threads running in parallel. They might got assigned some robi CPU time or might not. And we are not able to determine this until we actually executed all of them until they passed this time, too. How this problem can be solved is described in the next chapter.

But first some remarks about the "static" modeling used: I use a cooperative multitasking model to simulate preemptive (and cooperative) multitasking on the robi. Each preemptive (or cooperative) robi thread is represented by one cooperative simulation thread. As already mentioned earlier, the robi threads can be influenced in several ways, like suspending and resuming them, letting them sleep, blocking them by semaphores, etc. All these features have to be covered by the model. I try to simplify the handling of all this by distinguishing only between two different functionalities, the indefinitely suspending of a thread (like the usual suspending or blocking a thread with a semaphore) and the finitely suspending (sleep). Timers shall be excluded from the model at the moment.

Following are two chapters about two different task-switching models. Of importance for the final simulator is only the second one. The first is more like an intermediate step I made while thinking about a solution for the problem. I include it anyway, since it describes what I have done and might be of interest for future simulator projects.

2.3.1 The non-discrete task-switching model

The idea to calculate the *virtual time* is based on an inductive model: the induction hypothesis is, we have N threads that just entered a RoBIOS call and we know for every thread the *virtual time* when its last program segment execution started and the consumed CPU time for this segment. Now the induction step, we can determine the current *virtual time* for exactly one single thread (how to do this - later), then we store the *virtual time* of this thread as the start time of the following segment and execute this segment until an other RoBIOS call is made then we determine the CPU time, used for the execution of this segment and end up with a fulfilled induction hypothesis and we can start over. The induction start is easy, the *virtual time* of each thread is 0 and the CPU time for the first segment is simply measured.

Now I still owe the reader an explanation how to determine the *virtual time* of the mentioned *one* thread and how to find this thread at all. We want to find the thread that has the smallest *virtual time*, I will call it the last thread. We choose this thread, because its *virtual time* can not be influenced any more by the program executed in any of the other threads, so we can determine its *virtual time*. To simplify the model I assume the scheduled time slices are infinitesimal and the time cost of a task switch is zero, furthermore the finitely suspension of a thread and thread priorities are excluded. Then the *virtual time* of the last thread is its last known *virtual time* + the consumed robi CPU time of its last program segment times the number of currently running threads. We find this thread by finding the minimum of this values of all running threads. Then we set the current *virtual time* of the last thread to the just determined value. In case that the last entered RoBIOS call of the last thread changes the number of running threads, e.g. by suspending (indefinitely or finitely) or resuming a thread, we have to update the *virtual time* values and the robi CPU time values of all other threads, i.e. their current *virtual time* has to be set to the current *virtual time* of the last thread and their consumed robi CPU time has to be decreased accordingly.

We previously excluded the functionality of finitely suspension of a thread and this deficiency should me remedied now. The necessary modification to the above described algorithm is that the latest thread might be one, that was suspended and should be woken up by the scheduler now. Therefore we have to compare the smallest *virtual time*found by the old algorithm with the smallest wakeup time of all currently finitely suspended threads. In case, one of the finitely suspended threads wins, we have to update all *virtual time*, since this resuming obviously changes the number of running threads. This resumed thread is going to be executed next.

The timer functionality is quite easy to implement, since it is an interrupt and not a scheduled thread. Its consumed robi CPU time gets simply added to each threads *virtual time*. And since it is an interrupt handler, the timer function gets just executed until it returns. We probably have to forbid some of the RoBIOS calls to be executed within the timer interrupt (e.g. OSWait, OSYield, MTSem*).

Next thing to think about is the simulation of semaphores on the robi. The basic idea is to reduce the problem to this of suspending a thread. If one thread wants to get (perform the V operation) a blocked semaphore, it will simply mark itself as waiting for the semaphore in a FIFO and then get suspended. When the semaphore gets released, the first thread in the FIFO waiting queue will be resumed.

The described algorithm has two major disadvantages. First of all there is no modeling of proper scheduling time slices, i.e. not infinitesimal ones, and also no modeling of the cost (in terms of time) of a task-switch. Well, one might think this is an unimportant issue and is probably right. But second of all the performance of the simulation could get quite bad because of a vast number of necessary task-switches in the simulator, e.g. we have two threads doing the same, just keeping on calling a RoBIOS function, which has negligible robi CPU time consumption; the algorithm would switch the current task after every call of this function, because the latest thread keeps on alternating after each RoBIOS call. Trying to solve these problems I thought about the idea of introducing simulated time slices and ended up with an almost completely different, but therefor simpler algorithm.

2.3.2 The discrete task-switching model

The idea is to directly simulate the robi scheduler, which is almost just the execution of the scheduler code itself. The RoBIOS scheduler has a fixed time slice length of a hundredth of a second, I will call this time $TS = 0.01s$ (for time slice) and a (preemptive) reschedule happens always at a multiple of a hundredth of a second. Opposed to the old version, we maintain 1) only one *virtual time* for the robi, instead of one for each simulated thread, plus 2) the time of the next task switch ($NTST$), which is simply the next multiple of TS larger than the *virtual time* of the last task switch, plus 3) the not yet assigned, but already used, robi CPU time for each simulated thread ($NYAT$).

Given any situation during the simulation, we have one currently scheduled thread, whose robi CPU time consumption increases the *virtual time* of the robi. As soon as this thread enters a RoBIOS call and the robi's current *virtual time* at this date is greater or equal to $NTST$, we set the $NYAT$ of the current thread to current *virtual time* $- NTST$ and reset the *virtual time* of the robi to $NTST$. Then we find the next task to be

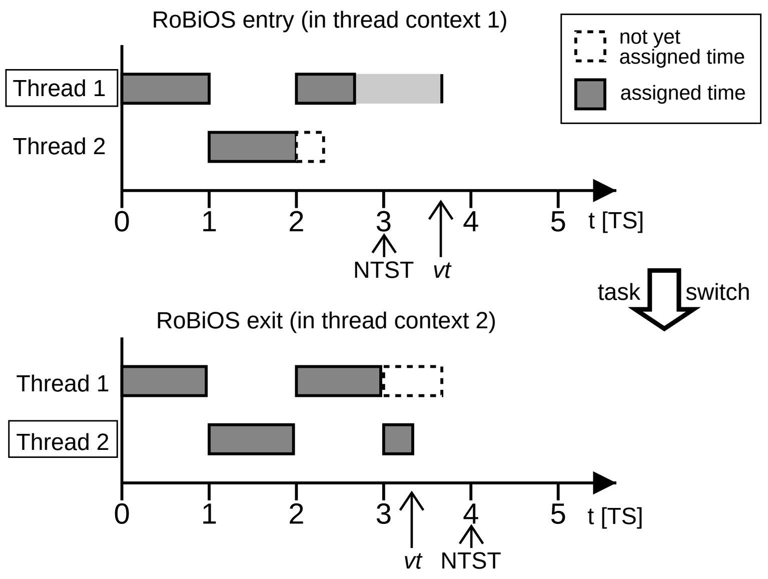


FIGURE 2.3: The *virtual time* handling during a (non-virtual) preemptive task switch

executed, using whichever, possibly priority based selection we like. We update the $NTST$, in this case it's just $NTST+ = TS$. We increase the *virtual time* by the new threads $NYAT$ and if the *virtual time* is less than $NTST$, we perform the context switch and thereby go on (see figure 2.3). But if the $NYAT$ of the new selected thread is larger than TS then the *virtual time* $>= NTST$, hence we don't need to perform the task switch, we merely perform a "virtual" task switch, meaning, we only update all *virtual time* related variables in the same way, as described above, i.e.: $NYAT = virtual\ time - NTST$ (complying with $NYAT- = TS$), $virtual\ time = NTST$ (complying with $virtual\ time+ = TS$) and $NTST+ = TS$. We keep on repeating this (selection of new thread plus performing virtual task switches) until $virtual\ time < NTST$ and we actually perform a real task switch.

To support finitely and indefinitely suspension of threads and timer-irqs we need to complement the algorithm. The indefinitely suspension and resumption of a thread is very easy, we simply mark the thread as suspended, so that the selection algorithm will skip it or unmark it for resumption respectively.

To implement the finitely suspension, we maintain a wakeup time for each thread and when this time is smaller than the current *virtual time*, the thread is executable, otherwise we skip it in the selection algorithm.

The timer-irq feature needs some more work to be supported. The timer-irq handlers get executed on a multiple of TS like the preemptive task switching, in fact the preemptive task switching mechanism is realized on the robi with one additional timer-irq handler. To accomplish this functionality we need two things, a list of all timer-irq handlers that should be executed and a variable, that stores the *n*ext timer interrupt time $NTIT$. If the *virtual time* gets greater or equal to $NTIT$ the timer-irq handlers in the queue have to be executed. For easier understanding, we treat the two cases of the preemptive and the cooperative multitasking mode separately.

In the preemptive mode the queue has to be processed whenever a task switch is performed, either a real or a virtual one. After the *virtual time* is set to $NTST$, which equals $NTIT$ in this case, it is sufficient to execute all timer-irq handlers once, thereby keeping track of the consumed robi CPU time and add this time to the *virtual time* before the we go on with the real or virtual task-switch. At this point we can easily check, if the timer-irq handlers have consumed more than TS seconds, which would mean we get an other interrupt while the handling of the old one is still in progress. (If this happens in the current implementation, the simulation continues in an undefined state in terms of the *virtual time*.) During execution of the irq handlers we have to set a special flag, that indicates the irq handler context for any RoBIOS function call made by the handler, so that we don't try to make, for example, task switches inside the irq handler.

When the mode is cooperative multitasking, it is a bit more difficult. On each RoBIOS function entry it has to be checked if *virtual time* $>=$ $NTIT$. Since the robi CPU time consumed by the last executed program segment could be a multiple of TS, it is possibly necessary to execute the irq handlers more often than once viz for each multiple of TS between $NTIT$ and *virtual time*. After each execution of the irq handlers $NTIT$ gets increased by TS. Before the execution of the irq handlers the *virtual time* has to be set back to $NTIT$, so its current value has to be saved and since the execution of the handlers consumes robi CPU time, too, it also has to be corrected, which means increased by the amount of time the handlers consume. At this it has to be minded that the handlers actually have to be executed for each multiple of TS until $NTIT$ is greater than the *c*orrected *virtual time*. This could be illustrated with an example: let the *virtual time* on the entry of a RoBIOS function be $NTIT' + TS/2$ and the execution of the timer-irq handlers takes $TS/2$ seconds, then the handlers have to be executed twice, the first time with the *virtual time* set to $NTIT$ and the second time with $NTIT' + TS$, since after the first execution the corrected *virtual time* is $NTIT' + TS$. After the second execution, the corrected *virtual time* is $NTIT' + 1,5 * TS$ and the new $NTIT = NTIT' + 2 * TS$.

A pseudo code fragment would look like this:

```
cvt = vt;
while( cvt >= ntit )
{
    vt = ntit;
    process_handlers(); /* this modifies vt */
    time t = vt - ntit;
    if( t > ts )
        warning();
    cvt += t;
    ntit += ts;
}
vt = cvt;
```

2.4 The structure of the RoBIOS functions

The following section summarizes the presented algorithms for the multitasking simulation and the synchronization and describes how and where they are embedded in the simulator implementation.

As mentioned in the introduction, the only way for the simulator to influence the execution of the robi programs or the robi threads, respectively, is through the implementation of the RoBIOS functions. The implementation of all RoBIOS functions has the same general structure, each is composed of three parts:

1. the entry part:
   This part is equal for each RoBIOS function. The first thing that is done here, is to stop the timer, that keeps track of the used host CPU time for the last program segment and to add the measured time multiplied with the earlier described factor to the *virtual time* of the robi, that called this function. If the robi is in preemptive multitasking mode and its *virtual time* $\geq NTST$ and the context is not a timer irq-handler then the timer irq-handlers are processed and all virtual task-switches and possibly one real task-switch is performed. For more detailed information about how the real task switch is performed, see the implementation details chapter (A). If the multitasking mode is not preemptive, there are no task switches performed, but the timer irq-handers are processed in case *virtual time* $\geq NTIT$.

   Now the current *virtual time* of this robi is determined and it has to be checked if a synchronization with the core is necessary, i.e. check if the time slot of the last synchronization time is an other than the time slot of the current *virtual time*. The synchronization has to be established before the execution of the functional part, since there happens the interaction with the environment, including the other robis.

2. the functional part:
   This is the actual implementation of the function, i.e. the function specific code. Details about the different functional groups and their implementation are presented in chapter 3.

3. the exit part:
   Two things have to be done in this part, first the increment of the *virtual time* by the amount of time that the function needs to be executed on the real robot, which has to be looked up in a table. The other thing is the restarting of the timer to measure the used host CPU time used for the execution of the next program segment. The exit part is also the same for every RoBIOS function.

2.5 The world model

This chapter is about the model used to specify the world or the environment in which the robis are acting. For all interesting simulations it is sufficient to use a finite, closed, two-dimensional plane. The only additional necessary entities are walls on this plane, so it is possible to create mazes or a soccer field, which is just the plane and some surrounding walls. The walls are not really three dimensional, since they have no width, all the same height and they are perpendicular to the ground plane. So all information, necessary for describing a world, is a set of two two-dimensional points, each point pair describes a wall segment. Other than the walls it is of course necessary to specify the start position of the robis inside the world. And for the soccer application, the ball (or the balls) have to be positioned as well.

There are two file formats supported to specify a world. The so called "world format", which is simply a plane text file, each line containing 4 numeric values, specifying the beginning and the end of a wall segment. Each robi is specified in a separate line, starting with the string "position" followed by three numeric values, the x-y position and the angle between the positive x-axis and the front of the robi (the orientation). The balls are identified by the string "ball", followed by the x and y-coordinate.

The other format is also text based and is called the "maze-format". It is supported, since it is intuitive and wide spread. Basically it is a graphical representation of the walls in ASCII using only the characters '−' and '|'. So the world can only contain either horizontal or vertical wall segments and they all have the same length. Of course several segments can be connected to a longer wall. This format is extended to support the specification of robi and ball positions, by introducing other characters like "lrudb".

More precise information about how to create a valid maze or world file can be found on the web pages [5] and [6].

2.6 The robi model

In this chapter I want to describe the way the robi is modeled in the simulator. There are different aspects of the model:

- geometrical model:
  This aspect describes what geometry the robi has in the context of the environment or the world, in which the robi is acting. This also includes the question of collision detection between the robi and the walls or other robis.

- dynamic model:
  This aspect is about the dynamics of the robi when it moves inside his environment. What movements are possible and what happens if the robi collides with obstacles?

- graphical model:
  This aspect is about the graphical representation of the robi within the environment as seen by other robis.

- error model:
  The error model describes which errors, occurring in the real world, like sensor errors, etc. are simulated and how.

In the following sections I will discuss these different models, but first of all I have to point out, that the used model was not properly developed. I started with a very simple model and had to extend that from time to time, to support new features or fix conceptual bugs (like robis tunneling through walls etc.). The result of this process is far from ideal and it should be considered to redesign this part of the simulator properly.

2.6.1 Geometrical model

The robi is simply modeled as a circle with a specified center point within the environment and a fixed, predefined radius.

Hence, the collision detection between a robi and a wall or between two robis respectively is rather simple: Two robis collide if and only if the distance between their center points is less than the sum of their radiuses. A robi collides with a wall segment if and only if either one of the wall segments end points is inside the robi circle or the plumbline of the robi center to the wall segment line traverses the wall segment line between the two wall segment confining points.

This simple model was used, because it is easy to implement and collision detection can be performed fast. The disadvantages are, that this simplicity could not be preserved for the ball-interaction model, it does not represent the real geometry very accurate and it is different from the graphical representation. The problem with the ball-interaction model is, that the robis should be able to actively manipulate the ball, like kicking it or "tripping" it. Therefore the circle-model of the robis is insufficient. So for this aspect of the simulation, I changed the robi model to a square instead of the circle.

So a more sophisticated geometric model for the robi would improve the simulation results and the overall simplicity by having one consistent geometric model.

2.6.2 Dynamic model

As already mentioned, the robi is a two wheel robot. Each wheel is driven by a separate motor. The angular velocity of the wheels can be controlled independently. With this setup the robi is basically capable of driving strait lines, circles or turning on the spot.

These movements can be parameterized with the standard $v\omega$-interface. The RoBIOS allows to access the drive directly but also provides the $v\omega$-interface. For the simulator, it was not of interest to implement a low-level access method, since all higher-level control algorithms use the $v\omega$-interface anyway.

The robis have no mass in the simulation, so we do not simulate bouncing of robis against walls or other robis. If a robi collides with an obstacle, it will simply be stopped. No mass also implies that the acceleration of a robi is infinite, if the robi program sets a new velocity, the robi immediately reaches that velocity. This is obviously not very realistic and could falsify the simulation results crucially, so this should be improved in a future version.

A rather simple approach to remedy this defect would be to introduce some kind of velocity slope: instead of jumping from one velocity to another, the simulation could blend from the old velocity to the new one.

2.6.3 Graphical model

The graphical model of the robi is of minor interest. It is only to provide some input for the camera simulation of the robis and the user interface monitor of the simulation. The generation of the image data in the camera simulation is based on a 3D graphics library, where the whole world, including the robis, the walls and the balls is modeled as a 3D scene. Therefore the robis need to be modeled as 3D objects. The implementation is based on the OpenInventor interface which allows the user of the simulator to specify its own 3D model for the robis. This model is completely independent from the geometrical model of the robis, but the shape and the size of the model should roughly resemble the robi circle to prevent graphic artifacts like robis penetrating walls or each other.

2.6.4 Error model

In the real world, there are above all two kinds of errors relevant for programs executed on the robis:

1. sensor errors:
   Sensor errors occur always when a physical device is used to perform measurements of some quantity, so for the distance giving PSD sensor. When measuring a given distance more than one time, the results will more or less differ. These errors have noise character.

2. odometer errors:
   The real robi's odometer is not perfectly accurate, since it relies on the measurement of the distance, covered by the robi's wheels and the presumption, that the wheels don't slide on the ground. Besides, the used algorithm is a numerical integration of the covered way and this numeric is another source of errors, in terms of rounding and discretization errors. This results in errors in the prediction of the robi's position.

In the simulation both types of errors are simulated by the addition of pseudo random numbers. The question is which statistical distribution the pseudo random numbers should be based on. An ad hoc answer to this question is the gauss distribution. For most measurement processes this is the real distribution. I simply used it, without an analysis of the real errors. Please see the next section for a justification of this lax policy.

The user can parameterize the produced errors by specifying the standard deviation of the gauss distribution, separately for each error variable. In the case of the PSD values, the error is just added to the correct value. To simulate the error in the odometer, I simply add a random number to the linear and the angular velocity when the program calls a drive command.

2.6.5 Remarks on the physical defects of the used model

The aim of the simulator was not to simulate physical processes in the environment of robis. It was rather to simulate programs written for these robis. Of course, the simulated robis should behave more or less like the real ones do, but as long as the executed program does not have a more sophisticated physical model about how the robis and their environment should behave than the simulator has, there is no need to implement a more realistic model.

Chapter 3

The functional part of the simulation

The RoBIOS functions can be divided in different functional groups, like the $v\omega$-interface related functions or the camera functions. In this section I will provide an overview of the different function groups and how their simulation is realized.

3.1 $v\omega$-interface

The $v\omega$-interface of the RoBIOS is a high level interface for controlling the two connected motors. It allows to specify a linear and an angular velocity component. Both may be negative or positive in a range depending an the physical properties of the system. The RoBIOS interface allows even more: it is possible to specify a drive command, consisting of the v and ω velocity and a time after which the robi should automatically stop. Instead of the time, usually the equivalent value of the distance which is to cover, is specified. In case the command is a turn command, where the linear distance is always zero, the angle, around which the robi should turn, is given. After initiating such a drive command, the program can ask the RoBIOS if the drive command is done, i.e. the desired distance is covered and the robi has stopped. There is also a blocking function to wait for the last drive command to complete and a function to get the remaining distance. The implementation of these commands on the real robi depends on an internal odometer. This odometer functionality can be utilized through the RoBIOSes interface as well. It allows to specify an arbitrary position in an x-y-α-coordinate system and to retrieve the updated position at a later time.

I thought about two possible approaches for the calculation model used in the simulation of the robi dynamics. The first is a time step based, integration method, where the robi's dynamic is represented in a state vector containing the current time t, the position p, the velocity v (and possibly the acceleration). The state at the next time step $t + \Delta t$ would be calculated by the equations of motion defined by the current state, evaluated at the time Δt. This is a fairly simple and flexible method but has the disadvantage, that rounding errors accumulate over time.

The other approach is similar, it can be given an account of an extension. Instead of updating the position on every time step, it is updated only when a new velocity is set, i.e. when a new $v\omega$-command is executed by the program. The state vector in this solution consists of: 1) the time t_1 when the current $v\omega$-command was initiated, 2) the position p, the robi was at, at the time t_1, 3) the v and ω component and 4) the time t_2, when this $v\omega$-command is done. Please note, that t_2 might have the value ∞. This information allows to define the equation of motion, describing the robi's movements. In a simplified notation this equation is just $position(t) = p + v\omega(min(t, t_2) - t_1)$, describing the position of the robi at an arbitrary time $t \geq t_1$.

With this solution, there is no accumulation of rounding errors, as long as the velocity is not changed. For example, if the robi should drive a circle, with this method, he will end up exactly where he started.

When a new $v\omega$-command is set at the time t', the state vector has to be update. The new value for p is the position of the robi at the time t' based on the old sate, the new time t_1 is t' and values for t_2, v and ω are just taken over.

In case the robi collides with an obstacle, the time of the collision is calculated and the velocity is set to 0, hence, the robi stops directly in front of the obstacle.

The implementation of the odometer functionality is based on the same algorithm. There exists a separate instantiation of $v\omega$-state information for the robi internal odometer. If the robi program initializes its internal position, this position is used as the start position p. Every subsequent call of a drive command updates both instantiations of the state information. The get function for the position calculates the position at the current *virtual time* , based on the internal odometers state.

3.2 PSD sensors

The Position Sensitive Detectors (PSD) use infrared beams to measure distance. The EyeBot controller can handle up to 8 connected PSD sensors. Usually a robi has three sensors connected, on pointing to the front, the others to the left and right. They can be operated in two different modes: the single measuring mode or the continuous measuring mode. They take about 60 ms to perform one measurement. In the continuous measuring mode, they start the next measuring automatically when the last one is done. The RoBIOS interface allows to initialize and release the sensors, to start and stop a measurement (cycle), to check for a valid value and to get the value.

The in the simulation it is possible to place the sensors at arbitrary positions relative to the robi center. This is specified in a description file for the robi. The value returned by a sensor is calculated by intersecting its “measure beam” with every wall segment and every other robi. The nearest intersection with any obstacle gives the distance value. As mentioned, a single measurement takes about 60 ms on the real robi, but the function to get the value is much faster, it simply returns the last measured value or an undefined value, in case the first measuring cycle is not yet complete. Thus, in the simulation a new value is only calculated everytime the world state gets updated, i.e. every 50 ms simulated time. There is no differentiation in the implementation of the single measurement mode and the continuous measurement mode, since the values are calculated only when they are demanded by the get function anyway.

3.3 LCD display

The LCD of a standard EyeBot controller is a 128 pixel width and 64 pixel height black/white display. The RoBIOS provides three ways to manipulate the content of the display:

- drawing characters or text
- drawing simple shapes like lines and filled areas
- fill the display with an image, e.g. an image grabbed from the camera

The simulator maintains two buffers for each robi, representing the contents of the display. One is a text buffer, where the text, to be printed, is stored. The other buffer stores the image information in a pixel based bitmap. But in contrast to the real LCD the image buffer contains colored pixels, since this requires even less effort to be implemented on the host machine and the additional information can be used for debugging purposes.

In case the user wants to see the content of the display, he can enable the console window in the simulator for that robi.

3.4 Key input

The keyboard of the EyeBot controller has 4 keys. The RoBIOS allows to read the keyboard buffer non-blocking or to wait for a specific key or any key to be pressed.

To make a keyboard input, the user has to display the robi console window and focus this windows, then all key strokes for the keys '1' to '4' from the host machines keyboard are sent to the simulated robi's keyboard buffer. The implementation for the wait-function is rather simple: for the non multi threaded case, I used a busy loop just waiting for the keyboard input from the user, in the multi-threaded case, the wait routine reschedules until a key is pressed.

3.5 OS, multithreading, timers and thread synchronization

The "OS" group contains some simple functions for retrieving information like the ID of the robi, its name or the current time. There implementation is strait forward.

The other functions build the core functionality of the simulator. Their implementation has been discussed in detail in chapter 2.3.

3.6 Camera

The EyeBot controller supports the control of one or more digital color cameras connected to the controller. The camera interface in the RoBIOS version 4.5, which was the current version when I wrote the simulator, supports only one camera. The interface is fairly simple, just an initialize and a release function and two functions for grabbing an image, one to get a grey-scale image, the other to get a color image. Both grabbing functions are blocking, i.e. they return after the new image is fully grabbed. There is an other function as well, for setting camera parameters like brightness and contrast, but it has no relevancy for the simulator.

The image generation is based on the same 3D library like the user monitor. The only difference is that an offscreen renderer is used, instead of displaying the data to the screen. The renderer gets an OpenInventor scene graph as input. This scene graph contains the graphical representation of the world, containing of the floor and the walls, as well as the balls and robis at their current position. To get an image from the point of view of the robi, each robi has to use its own camera position for the rendering. The used position is the robi's current position plus an offset, the relative position of the camera to the robi's center.

The soccer robot has the camera mounted on a pan-tilt unit controlled with servos. Since the simulator does not simulate the servos directly, I had to introduce a new function to the set of RoBIOS functions, provided by the simulator. This functions sets the pan and tilt angle of the camera. It is likely that a program for the real robot abstracts the servo interface in the same way, so it is no problem to incorporate this extra function in the program during the simulation. In case the reader is wondering: yes, this undermines the demand, to be able to use exactly the same source code for the simulation and the real world usage.

So the steps necessary to generate an image from the point of view of the robi's camera are: calculate the camera position using the robi's current

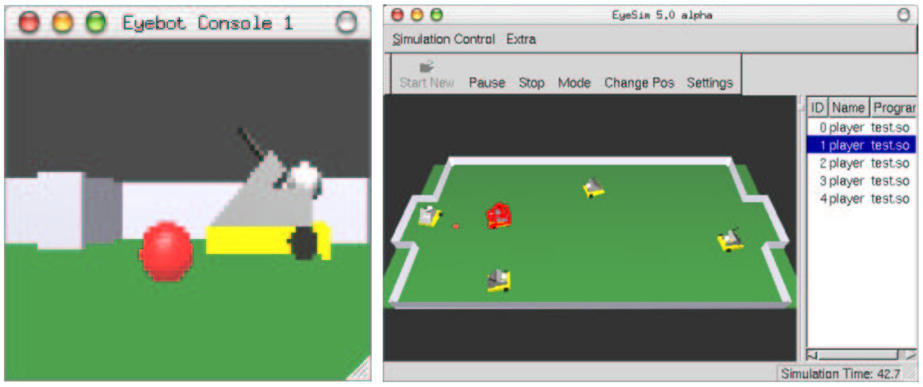


FIGURE 3.1: An example of an image, generated with the simulated camera (left); The sceene from a birds view, the active robi is marked red (right)

position, the camera offset and the current pan-tilt values then generating an OpenInventor camera, with this coordinates and finally call the offscreen renderer. For an example of an image, generated with the offscreen renderer, see figure 3.1.

As hinted before, the new RoBIOS camera interface supports more than one camera. This is yet to be implemented in the simulator, but it should be no problem, since I have already prepared it. The only missing part is the interface itself, because the exact signature was not decided then.

3.7 Radio communication

As mentioned before, the robis are capable of communicating with each other through a radio communication interface. This interface is rather simple: it allows to send a message to a specific robi or to all of them, to check if a message is in the queue and to receive a message. The receive call is blocking in case the message queue is empty. The messages are just variable length data blocks, containing arbitrary binary data.

To realize this feature in the simulator, each robi client maintains a message queue. Since the single robi clients do not know each other, they have to send the message to the core. The core dispatches the message either to the specified robi client or to all of them, if the message is a broadcast message. The message is then stored in the message queue until the robi program calls the receive function.

For simplicity (and because I had no time to make it better) I implemented the receive function as a busy loop, exactly like the key wait function, see section *Key input* (3.4).

3.8 RS232

The EyeBot controller has a RS232 interface for serial line communication with a PC. This link is used for firmware updates and the transfer of the program data. This functionality is not simulated since the firmware equivalent is the simulator itself and the program binaries are specified in the simulation description file. But the serial line can also be used to transfer data back to the PC, e.g. to get some debug information or result of the program like a generated map of the robi's environment.

This is simulated by simply creating a text file on the host machine and adding every character passed through the RS232 interface to that file. The "get" function, to receive data form the PC as well as the control functions to set transfer speed, etc. are merely dummy implementations.

3.9 The rest

The RoBIOS contains other functions not mentioned so far. These are functions related to audio signal emitting, low level motor access, infra red communication, electronic compass evaluation, servo control and others. All of these functions have been classified as not important for simulation. Their implementation is merely a stub or dummy implementation. If a value has to be returned, then the "no error" value is used, thus allowing the user to simulate programs using these functions without the necessity to modify them. This, of course, works only for programs that don't really rely on these functions.

Chapter 4

A brief manual

I want to give a brief introduction to the usage of the simulation software, interesting for someone, who wants to use the simulator.

To be able to simulate a robi program in the simulator, a simulation environment has to be set up. The simulator needs certain information about the robis to be simulated, the virtual world, in which the simulation takes place and some other parameters like the simulation-to-real-time ratio and the program(s) to be simulated. All this information has to be provided in different files.

- simulator configuration (eyesim.ini file):
  This file contains settings for global parameters in the simulation. These include display parameters (display update rate, background color, etc.), the error model parameters (see chapter 2.6.4), the parametrization of the ball simulation (ball radius, friction and reflection coefficients) and the maximum reach of the PSD sensors. All these parameters may be overwritten in the simulation description file. Some of the parameters can be changed online during the execution of the simulation.

- simulation description (.sim file):
  This file describes the simulation, i.e. what world should be used and which robis are involved. This is done by merely providing the path for the robi description file and the world description file.

- robi configuration (.robi files):
  This file provides the complete description of a robi, including a name, the size, the maximum linear and angular velocity, further more the specification of all PSD sensors and cameras with their relative position. It also references two other files: the robi's 3D-model and the program code. In the current (Linux) implementation, the program code has to be provided as a shared library (.so file), please see the implementation details for more information on this topic.

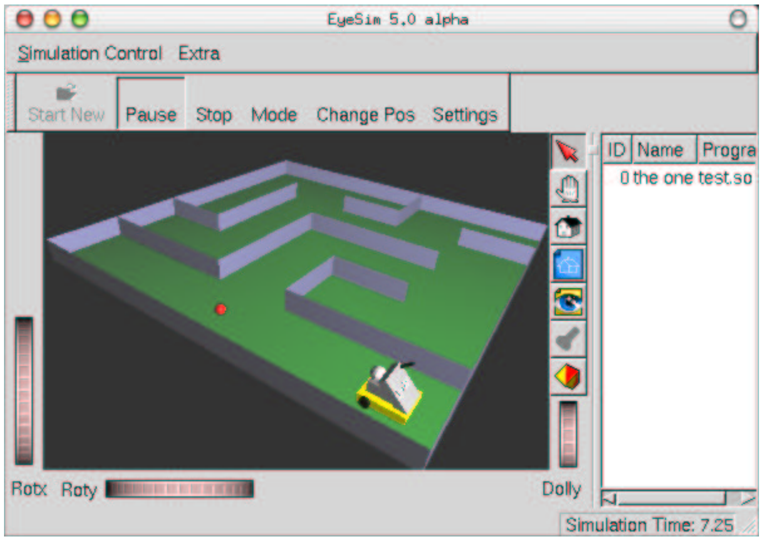


FIGURE 4.1: The main window of the EyeSim program

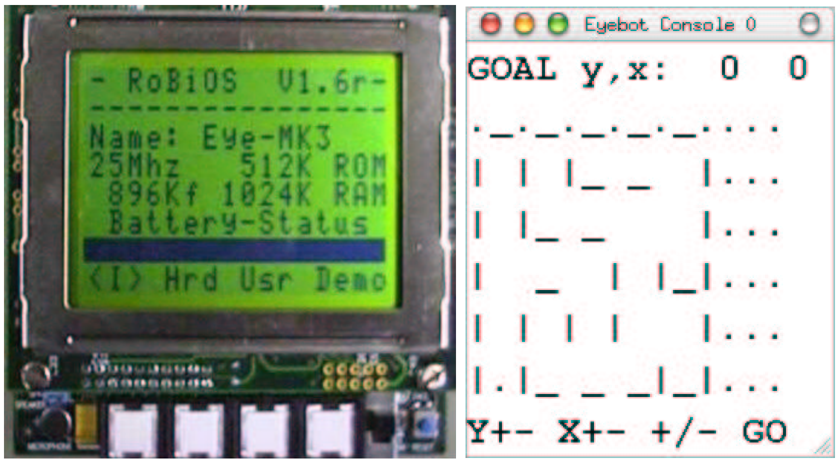
- 3D-model for the robi (.iv file):
  This file has to be in the OpenInventor file format (VRML 1.0). There is a relatively basic model provided with the simulator demos but the user may provide its own model. The unit to use for the model is 1 meter.

- world description (.wld/.maz file):
  There are two file formats supported: the so called "world format" and the "maze format". Please see the "The world model" section for more details about these formats.

There are sample files for the simulator, the simulation and the robi description to be found in the appendix. These samples provide a description for all possible parameters and sane default values.

Before the executable can be started, the user has to make sure, that the environment variable LD_LIBRARY_PATH contains the path to the shared library file(s) with the program(s) to be simulated. This may be the current directory ("."). The simulator configuration file (eyesim.ini) has to be placed in the current directory. The simulator is started by executing "eyesim", an optional parameter is the name of the simulation description file.

The simulator main window is shown in figure 4.1. It is composed of two main elements, the 3D viewer on the left and a list with some information about all robis in the simulation on the right. The viewer is a standard OpenInventor examiner viewer, it initially shows the world from a birds view. The perspective can be changed with the mouse. The list with the robis show their ID, their name as specified in the robi description and the file name of the program they are executing.



```
- RoBiOS  V1.6r-
----------------
Name: Eye-MK3
25Mhz     512K ROM
 896Kf 1024K RAM
 Battery-Status

<I> Hrd Usr Demo
```

```
GOAL y,x:   0   0
._._._._._....
| | |_ _   |...
| |_ _     |...
|  _  | |_|...
| | | |    |...
|.|_ _ _|_|...
Y+- X+- +/- GO
```

FIGURE 4.2: The EyeBot controller with LCD and buttons (left); The EyeSim console window, representing the LCD (right)

The simulation can be paused and stopped and a new one can be started with the appropriate buttons.

The robis can be selected in the list, a selected robi is marked with a red bounding box in the 3D viewer. Selected robis can be manually moved with the ChangePos dialog. A double-click on a robi in the list opens its console window. In case there is only one robi in the simulation, this window will be opened automatically. An example of the console window can be seen in figure 4.2.

The console window is the graphical representation of the robi's LCD. In the last text line is usually shown the menu of the program. The keyboard input for the robis (see bottom of figure 4.2 left) can be generated by focusing the console window and pressing they keys '1' to '4' on the host keyboard.

There is another dialog, the settings dialog, to change global settings, like the error model parameters and the user monitor related parameters.

The GUI of the simulator can be considered to be subject to further improvements. The most obvious improvements would be:

1. provide more information about the current state of the simulation, like all the current sensor readings of the robis, the exact positions of the robis and the balls

2. implement a graphical representation of the robi's current drive command, e.g. establish some sort of an arc in the 3D-viewer, pointing from each robi to its current driving destination

3. improve (restrict) the use of the OpenInventor examiner viewer - it makes no sense to be able to change the viewport in a way that you see nothing anymore

4. add some graphical representation of the robi keyboard to the console window, enabling the user to use the mouse as an input device, too

Appendix A

Implementation details

For the case, one is interested in having a closer look to the implementation of the simulator, I decided to collect some information and explanations regarding this issue here.

I want to give some more details about the code structure of the core and the robi clients. As described earlier the core functionality is provided by a class called Core. The main class representing the client is the LocalRobi. LocalRobi because there exists also a GlobalRobi class on the core side with one instance for each robi in the simulation, handling state information like current position, etc. The message passing interface is realized with two classes, the Core2ClientMessages and the Client2CoreMessages. Both clasess are interface classes describing the set of messages that can be passed between the core and the clients. Their implementation is responsible for delivering the messages. This abstraction allows a simple separation of the core and the client code to be executed in their own processes, since the only link is the message interface. In the thread based implementation, these classes merely have to ensure the thread safety when several robi clients send messages to the core simultaneously, but other than that they simply call their corresponding methods. For simplicity this interface is currently not complete, some RoBIOS function implementations have direct access to the world state information, which is maintained by the core. These are the PSD and camera related functions. To realize the process based implementation, it would be necessary to replicate this information on the client side and add messages to the interface to be able to synchronize the data. This has not yet been done because of a lack of time.

As described in the multitasking section, the used model for simulating both, preemptive and cooperative multitasking, is a cooperative model, i.e. (host) task switches are performed only explicitly. A C-library that supports coperative multitasking is described in [3]. However, since there was no platform independent library available (including Windows support), the implementation of the cooperative threads uses preemptive threads (see [4]).

To accomplish this "simulation" of cooperative threads with preemptive threads, each thread has assigned one semaphore with an initial value of 0 and the first thing, that the thread routine does, is locking itself by waiting for the semaphore. Starting the multitasking of one simulated roboi means to resume the first of its threads by increasing the thread's associated semaphore. When a task-switch has to be performed, the one running thread starts the next thread by releasing its semaphore and then blocks itself by waiting for its own semaphore. To easily illustrate this, one can think of a conservative petri-net, composed of a complete graph with only one token; switching the task means passing the token to an other place.

Appendix B

Configuration files

B.1 Simulator configuration (eyesim.ini)

```
# this is the global ini file for the EyeBot simulator 5.0. every value
# has buid in standard value, that can be overwitten in this file. it is
# also possible to specify any of these parameters in the simulation
# description file (*.sim), which will overwrite it again.

# the format is pretty obvious...

/** The rate with which all the user interface displays should be
 * updatd. The unit is one second, i.e. it is the reciprocal of frames
 * per second. The default is 0.2, which means at least 5 fps. */
displayUpdateRate 0.2

/** The ratio of simulated time to real time, e.g. a value of 1.0 means
 * the simulation runs in "real time", a value of 2.0, that the
 * simulation runns twice as fast. A value of 0.0 means "as fast as
 * possible". */
simulationToRealTimeRatio 1

/** This value is the largest distance, that a robi has from an object
 * after it collided with it, i.e. the ideal value would be 0, BUT this
 * is not feasable. Default value will be set to one mm, since the
 * accuracy of all distance giving sensors is just one mm. [m] */
# collisionDistanceAccuracy 0.001

/** these are the parameters for the error model, which simply adds a
 * random number of a gauss distribution with a standard deviation
 * like specified to the corresponding value. The default value is 0,
 * which means no error is added at all. [m]/[] */
psdStandardDeviation 0.0
vwLinearDistanceStandardDeviation 0.0
vwRotationalDistanceStandardDeviation 0.0


/* this is the deceleration factor of the ball movement, the balls velocity
 * decreases by Ball::friction * t, where t is the simulated time */
Ball::friction 0.03

/* the ball will stop, when he reached this threshold velocity, so
 * possible values are greater or equal to 0. */
Ball::thresholdVelocity 0.0
```

```
/** what could that possibly be? the default value is the radius of a
 * golf ball in [m] */
Ball::radius 0.02125

/* the default values are the usual collision coefficients for an
 * elastic impact (1, -1). the first coefficient gets multiplied with the
 * component paralell to the wall, the second with the component
 * perpendicular to the wall, i.e. the second component hast to be
 * <= 0. */
Ball::wallReflectionCoefficients 0.8 -0.5
Ball::robiReflectionCoefficients 0.0 -0.1

/** the ratio between robi cpu speed and host cpu speed, e.g. 10 means
 * the host cpu is 10 times faster than the robi cpu. This value is
 * not really of any use at the moment, since the accuracy with which
 * the host cpu time can be measured is not sufficient. */
# robi2HostCPUSpeedFactor  10

/** the maximum distance the psd sensors can measure, values greater
 * than maxReach are mapped to the constant PSD_OUT_OF_RANGE. */
PSD::maxReach 2.0

/** the background color used for the generation of the robi camera
 * images, given as (r,g,b) in float format with values between 0 and
 * 1. */
backgroundColor  0.3 0.3 0.3
```

B.2 Robi description (*.robi)

```
# robi description file for the EyeBot simulator EyeSim 5.0
# by Axel Waggershauser, UWA 2002

# inspired by the parameter file for the old simulator by
# Thomas Braunl - Ghee Ong - Nicholas Tay, UWA 1998

# the format: valid lines contain a keyword followed by a list of
# parameters seperated by white space. every line not starting with
# a keyword is ignored.

# every parameter describing a length is specified in milli meter
# every parameter describing a angle is specified in degree


# the name of the robi, the used string is the rest of the line
# behind the keyword without the surrounding white space
name player

# robot diameter mm
diameter 160

# max linear velocity in mm/s
speed    600

# max rotational velocity in deg/s
turn     300

# the file name of the shared library containing the program to be
# executed for this robi
program  test.so

# the file name of the OpenInventor model used for this robi
```

```
model    robi.iv

# axis is the distance between the center of the robi and the center of
# the robis axis, e.g. a value of 0 means the axis is in the center of
# the robi, if no value is given, it is set to 0.
#axis     60

# psd sensor definition:
# "psd", name, id-num, relative position to robi center, deviation angle
# id-num has to match file "hdt.h"
psd PSD_FRONT     -200 80 0 0
psd PSD_LEFT      -205 0 80 90
psd PSD_RIGHT     -210 0 -80 -90
psd PSD_BACK      -212 -40 0 180
#psd PSD_FRONT2    -211    5  80 -5
#psd PSD_LEFTDIAG  -203   45  80  0
#psd PSD_RIGHTDIAG -204  -45  80  0


# color camera sensor definition:
# "camera", relative position to the robi center (x,y,z), default
# pan-tilt-anlge (pan, tilt), maximum image width and height in pixel
camera 80 0 50 0 0 80 60
camera 80 10 50 0 0 80 60
```

B.3 Simulation description (*.sim)

```
# world description file (either one maze or one world)

# world test.wld
maze test.maz

# robi description file
robi test.robi
```

Bibliography

[1] Th. Bräunl, B. Graf, Small Robot Agents with On-Board Vision and Local Intelligence, Advanced Robotics, vol. 14, no. 1, 2000, pp. 51-64 (14)

[2] Th. Bräunl, Scaling Down Mobile Robots - A Joint Project in Intelligent Mini-Robot Research, Invited paper, 5th International Heinz Nixdorf Symposium on Autonomous Minirobots for Research and Edutainment, Paderborn, Oct. 2001, pp. (8)

[3] Ralf S. Engelschall, Portable Multithreading - The Signal Stack Trick For User-Space Thread Creation, USENIX Annual Technical Conference, June 18-20 2000, San Diego, USA.

[4] GNU Common C++, GNU Common C++ is a C++ framework offering portable support for threading, sockets, file access, daemons, persistence, serial I/O, XML parsing, and system services, initially started by David Sugar and Daniel Silverstone. http://www.gnu.org/software/commonc++/CommonC++.html

[5] Th. Bräunl, EyeSim - Maze Format, http://www.ee.uwa.edu.au/ braunl/eyebot/sim/maze.html

[6] Th. Bräunl, EyeSim - World Format, http://www.ee.uwa.edu.au/ braunl/eyebot/sim/world.html

List of Figures