Collaborative Robotics with Lego Mindstorms

Master Thesis of Andreas Junghans
Department of Computer Science
Karlsruhe University of Applied Sciences, Germany

03/01/2001 to 07/31/2001

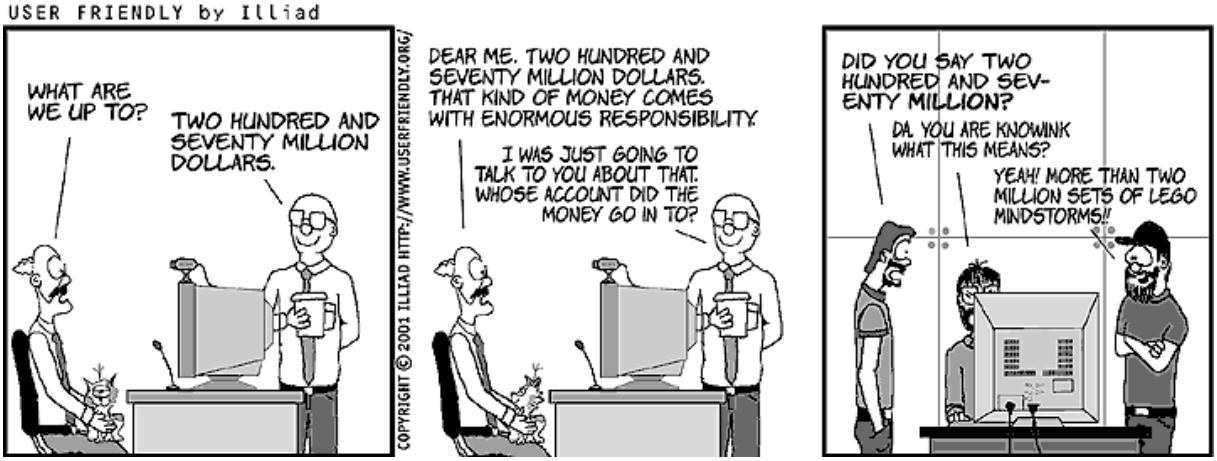
Supervisor: Prof. Dr. Peter A. Henning
Co-supervisor: Prof. Heinrich Herbstreith

Declaration

I have written this thesis independently, solely based on the literature and tools mentioned in the chapters and the appendix. This document – in the present or a similar form – has not and will not be submitted to any other institution apart from the Karlsruhe University of Applied Sciences to receive an academical grade.

Karlsruhe, 10th of August, 2001

(Andreas Junghans)



http://ars.userfriendly.org/cartoons/?id=20010723&mode=classic

Contents

List of Figures

List of Tables

1 Introduction

1.1 Document conventions

Throughout this thesis the following conventions are used:

- Important terms are *italicized* when first used.
- Within a glossary entry, *italicized* terms indicate that these terms have an entry of their own.
- All code fragments are formatted in `typewriter style`.
- Names of functions are marked with trailing parentheses. Example: `turnRight()`.
- References are made by sections numbers rather than names.
  Example: see section 1.1.
- Figures and tables are numbered on a per chapter basis.
- References to figures and tables are made without page numbers, except the figure or table in question is more than 3 pages away from the reference.

This document has been typeset using LaTeX2e in the MiKTeX 1.20e distribution.

1.2 Subject of this thesis

The intention of this thesis was to show different possibilities how two or more autonomous robots can work together to solve a problem – e. g. climbing stairs or finding their way through a labyrinth. However, the core of my studies was making the robots play soccer - and this took much longer than planned ...

There is already one very successful Mindstroms soccer project described in [LP00] by Henrik Hautop Lund and Luigi Pagliarini which is used for the Junior league of the annual RoboCup event [Rob]. This is a great piece of work and far more advanced than I got in this thesis! However, there are two main caveats:

- Lund and Pagliarini have used special balls, emitting infrared light of the frequency used by the Lego light sensor. This way, most of the ball finding problems have been solved – but without this specialty, the robots cannot play at all!
- The robots have been equipped with three light sensors and other parts not contained in a standard Robotics Invention Set (see section 2.1.1.2).

Both make it difficult to copy the setup for own experiments. In contrast, I wanted to develop an environment that works with just a few standard components - more precisely one Robotics Invention Set per robot and an additional Vision Command overhead camera. I also used a few wooden bars for the arena, but that doesn't count ...

Now, after 5 month, I have robots that can play fairly well (especially **without** an opponent), and that are great fun to watch (especially **with** an opponent). Due to the numerous problems I had to solve (which you can read about in the following chapters) there was not enough time for the other projects I had intended. Nevertheless, I hope that I can offer some useful tools to the Lego Mindstorms community as a result of this work. All software developed for this thesis is available at [Juna].

Have fun with it!

2 About Lego Mindstorms

2.1 Hardware and firmware

2.1.1 The Robotics Invention System 1.5

The core product of the Lego Mindstorms series is the *Robotics Invention System (RIS)*. It is intended to teach children and adults the basics of robotics using familiar Lego bricks. With the RIS you can build robots that move and react to inputs from their environment, e. g. touch and light. The robots' programs are written on a host computer (see section 2.2), downloaded to the robot via an infrared connection, and then executed autonomously. The latter is probably the most fascinating about Mindstorms – no cables or any other connection to a stationary computer is required for the robots to move around.

The current version of the RIS is 1.5 which includes an additional manual and online videos for novice users that were not present in 1.0.

2.1.1.1 The RCX and the Lego firmware

The most important part of the Robotics Invention System is the RCX. This is a special Lego brick with an integrated Hitachi H8/3292 microcontroller (running at 16 Mhz), three sensor inputs, three motor outputs, an IR transceiver, an LCD, and four control buttons. A simple speaker for sound output is also integrated. The RCX requires six batteries of type AA (Mignon) to run. If you have seen how powerful the motors are, you understand why ...

Besides low-level hardware access routines, the 16 KB on-chip ROM of the RCX only provides basic communication logic to download a *firmware* that serves as the execution environment for any user-written programs. However, there are also tools to program the RCX directly (see [Cap00]). Additionally, the RCX contains 32 KB of RAM that is shared by the firmware and the actual programs.

The following pages describe the features of the RCX in the context of the current Lego firmware (Version 2.0). This version does not come with the RIS 1.5 but is available as part of the RCX 2.0 Beta SDK [LEG00] and also included with the Vision Command camera (see section 2.1.2). Most of the features are available in other execution environments too, but organized differently. The most interesting alternative to the Lego firmware is called *leJOS* [Sol] and provides a Java runtime environment. More information (including source

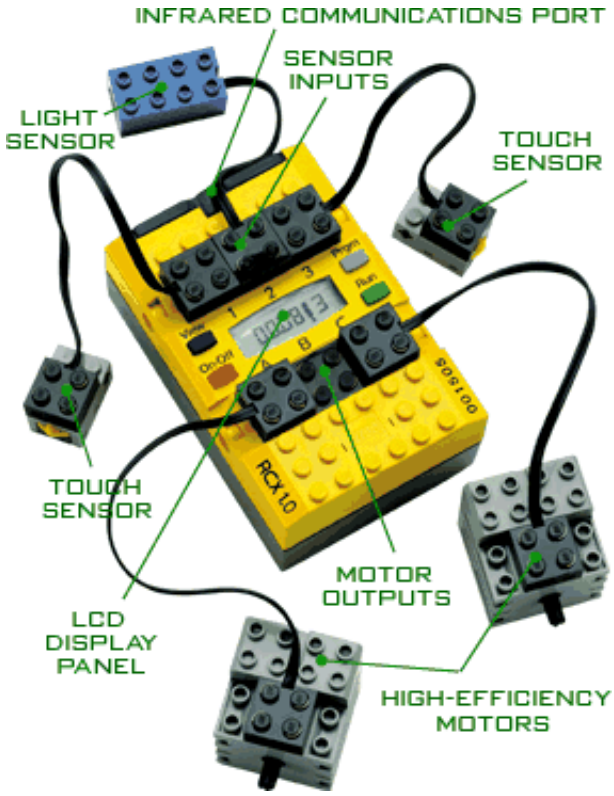


Figure 2.1: The RCX
(Source: http://mindstorms.lego.com)

code) can be found at http://lejos.sourceforge.net/. It should be noted that leJOS requires only 17 KB of memory while the Lego firmware consumes about 24 KB, leaving not much space for user programs.

Besides the RCX, Lego offers two other programmable bricks (short *P-bricks*) called *CyberMaster* and *Scout*. The latter is included in the *Robotics Discovery Set*. Of all P-bricks, the RCX is the most advanced.

Commands and programs The Lego firmware is basically a bytecode interpreter. A user program must be downloaded into one of five *slots* (using the IR interface) before it can be executed by the firmware. All bytecodes are documented in the RCX 2.0 Beta SDK [LEG00]. Besides downloading a complete program, individual commands can also be executed directly via the IR link.

All commands consist of an opcode (1 byte) and, if required, parameters (up to 5 bytes). The RCX secures every command transmission by performing a logical not operation on the command byte and sending back the result (except for bit 4 which has a special meaning). Downloads of firmware and programs are secured by simply adding up the bytes and performing a modulo operation on the result (mod 256 for programs, mod 65536 for firmware).

Tasks, resources, and events One of the outstanding features of the RCX (or rather the Lego firmware) is built-in multitasking. Up to 10 concurrent tasks per slot are possible. The priority of each task can be adjusted between 0 (highest) and 255 (lowest).

Since multitasking can lead to conflicts – e. g. when two tasks want to play sound at the same time –, the Lego firmware manages four resources: the three motors outputs and sound output. Each task can request access to these resources. Access is granted if no task of higher priority owns the resources in question. Access is denied (or taken away if granted before) as soon as a task with equal or higher priority requests the same resources. A task can specify a handler routine that is executed when it loses any of its acquired resources, so it can act appropriately. It should be noted that access control is not mandatory. If a low priority task uses motor commands without applying for access first, these commands are executed even if another task currently ”owns” the motors.

A task can also monitor up to 16 different events. These events are freely configurable, i. e. you can specify which event source to monitor and what type of event you want to receive. The possible event sources are:

- A physical sensor (touch, light, rotation, temperature, or self-made).
- A timer.
- A counter.
- An incoming IR message.

In addition to the event source, you must specify what type of change should trigger the event. For example, if you specify EVENT_TYPE_LOW, the event is triggered when a (configurable) lower limit is reached. The firmware supports 11 different event types, including clicks (a value goes from low to high and back to low), rapid changes, and incoming IR messages. To receive events, a task specifies an event handler that interrupts normal execution as soon as an event is triggered.

Sensor inputs As already mentioned, the RCX has three sensor inputs. You can connect one or more sensors to each input. If more then one is connected, you may or may not be able to differentiate between them (see [Ove99]), so the usual case is having one sensor on one input. Sensors are connected to the RCX with special cables that have small lego bricks with metal contacts at both ends. Some sensors (e. g. the light sensor) have cables where one end is hardwired to the sensor.

There are five types of sensors that can be used with the RCX (this number may grow with future firmware versions): touch, light, temperature, rotation, and generic. Generic means that, unlike the other types, the sensor signal is not preprocessed in any way. This is useful for self-made sensors. Sensor values can be used as event sources or queried directly.

Timers and counters The RCX has four timers with 10 ms resolution. Timers can be cleared (i. e. set to 0), set to a specific value, queried directly, and used as event sources.

Counters are similar to variables (see below). However, counters can only be incremented or decremented by one. What makes them interesting is that they can be used as event sources.

Variables, sub routines, and program flow control All variables provided by the Lego firmware are 16 bit integers. There are 32 global and persistent variables, i. e. they can be accessed from every task and sub routine and keep their values even if the RCX is turned off. Another 16 variables are local per task and can be used for temporary variables or parameter passing. It should be noted that the term variable is a bit misleading. The way they are organized and accessed makes them more similar to registers really. Variables can be manipulated using arithmetic operations (+, -, *, /), logical operators (`and`, `or`), and special commands for calculation of the sign and the absolute value.

For repeatedly needed parts of a program, the firmware provides up to 8 sub routines per task. Unfortunately, a sub routine is not allowed to call another sub routine.

The firmware facilitates flow control by conditional and unconditional jumps.

Motor outputs The RCX can drive motors through its three 9V outputs. Only one motor can be connected per output. The behaviour of the motors is determined by the mode, direction, and power attributes. Mode can be one of *on*, *off*, or *float*. On means the motor is turning with the current power setting, off means the motor actually breaks, and float lets it

Figure 2.2: The display and buttons of the RCX

move freely, but without turning on its own. The motor direction can be *forward* or *reverse*. The power of a motor is a constant between 0 (low) and 7 (high).

The RCX's three outputs are independent, so the attributes just described can be specified per motor. For example, a robot with one motor on each side can turn around by setting one to *forward* and one to *reverse* direction.

Motors can also be globally disabled, set to a specific direction, and limited regarding their maximum power. If a motor is diabled, none of the regular motor commands has an effect. If a motors is globally set to reverse direction, all direction commands are interpreted inversely, i. e. forward as reverse and reverse as forward.

Besides motors, other actuators can be connected to the outputs as well. For example, Lego offers a small "lamp" the brightness of which can be regulated through the output power. Naturally, directional settings don't make sense in this case.

Display and buttons The RCX contains an LCD with a programmable 4-digit section. It can be set to monitor a value (e. g. a timer) continously with only a single command. The display also shows the current execution state (program running/not running), the fill state of the datalog (see below), the number of the currently selected program slot, the state of the motors, and the state of connected touch sensors (small triangles that show up when a sensor is pressed).

The four buttons on the RCX are titled *On/off*, *Run*, *Select Pgm*, and *View*. The On/off button should be self-explanatory ... The Select Pgm button allows to cycle through the five program slots. The program in the currently selected slot (if present) can be started using the Run button. Finally, the View button allows to display the values of connected sensors in the 4-digit section mentioned above.

The IR transceiver Communication from the RCX to a host computer or another RCX is realized through the integrated IR transceiver. The supported baud rates are 2400 and 4800. Other settings that can be specified programmatically include sending power (long vs. short range), carrier frequency, and packet format. Details about the IR protocol can be found in [Pro99].

Besides uploads and downloads from and to the host, the firmware supports the exchange of simple messages between RCX bricks where each message consists of a single byte. Re-

ceived messages are stored in a 1-entry buffer from where the currently running program can read them. Another message can only be received when the program has cleared the buffer.

In [Ove99], Mark Overmars also shows a way to combine the IR transceiver and a light sensor to get a rough proximity sensor. Unfortunately, there is no other way of measuring distances, except with self-made sensors.

Miscellaneous In addition to the features already mentioned, the RCX supports sound generation. First, there are some built-in system sounds (like frequency sweeps up and down). The second way of producing sound is by specifying a frequency/duration pair.

A very useful feature for program debugging is the *datalog*. This is a memory area in which 16 bit integer values can be stored during program execution. Before the log can be used, it must be initialized by specifying the desired number of integers to store. After it is filled up (which is shown on the LCD), the log can be uploaded to the PC for examination. This way, you can, for example, look at the values a light sensor collected while the robot was moving around.

Finally, there is an integrated watch in 24 hour format. Probably its only useful purpose is being displayed on the LCD.

The software developer's view The RCX and the Lego firmware offer a mixture of high and low level commands that can be slightly confusing. On the one hand, there are features like tasks and event monitoring which allow for a high level of abstraction. On the other hand, variables and sub routines are very limited and make it difficult to structure a program. This continues right up to the development tools presented in section 2.2 which share both the benefits and constraints with the firmware.

What is really annoying about the current firmware version is its enormous size. If you subtract the 24 KB of the firmware from the available 32 KB, only 8 KB are left for own programs and the datalog. Also, performance can be an issue at times since all commands are interpreted.

2.1.1.2 Lego Technic parts and sensors

Most of the RIS consists of standard Lego Technic parts. Only the two included motors seem slightly different, and there are a few special parts some of which are shown in figure 2.3. Additionally, the RIS contains two touch sensors and one light sensor that can be connected to the RCX.

With more than 700 pieces, one RIS set already allows for rather complex robot constructions. If that isn't enough, one can always purchase additional Lego Technic sets or one of the Mindstorms Expansion sets (see the Lego Mindstorms Homepage [LEGa]). More and different sensors can also be purchased individually, most notably a rotation and a temperature sensor.

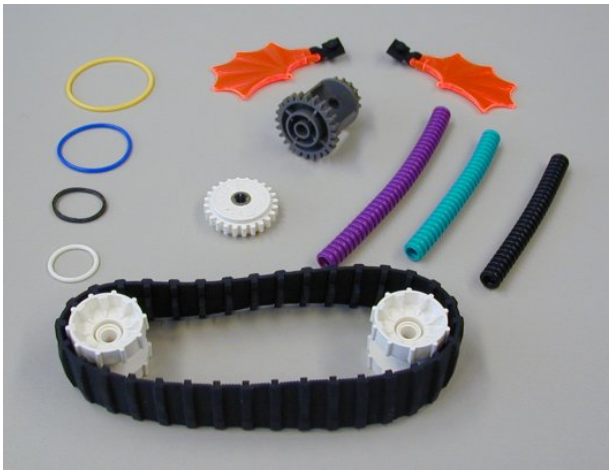


Figure 2.3: Special Mindstorms parts

Figure 2.4: Mindstorms IR tower

2.1.1.3 The IR tower

An important part of the RIS is the so-called *IR tower* which serves as a communication facility between a PC[1] and the RCX. It needs a 9V block battery to run and is connected to a serial port with a 9-pin cable.

Like the RCX transceiver, the tower has two range settings (near and far). With both the tower and the RCX set to far, I have been able to cover a distance of 1.6 meters! However, the tower quickly empties the battery with this setting.

In the 2.0 version of the RIS, at the time of writing only available in Japan, the serial port connection is replaced by the Universal Serial Bus (USB). The great advantage is that the USB provides enough power to run the tower without a battery.

[1] You can also connect it to other computers like the Apple Macintosh (see [Baub]), but all the original Lego software only runs under MS Windows.

Figure 2.5: Vision Command camera

2.1.2 The Vision Command camera

Lego offers a CCD camera connected to the USB port of a PC that can be used for light, color, and movement recognition. The camera can be connected to standard Lego technic parts and thus be mounted on a Mindstorms robot. However, the USB cable keeps the robot tied to the host computer. Although 5 meters long, this is not an option for wildly moving soccer robots since they would sooner or later get entangled.

The camera delivers still pictures and streaming video at a maximum resolution of 352x288 (Y channel) and 176x144 (U and V channels) respectively. The image quality matches that of average web cams. A small microphone is also built-in. There are only two controls: a focus ring and a button for taking still pictures.

The Vision Command set additionally includes 145 Lego Technic parts for different camera stands and attachments.

2.2 Software

2.2.1 The Robotics Invention System programming environment

The following sections are based on Version 1.5 of the RIS. At the time of writing, Version 2.0 is only available in Japan, but part of its functionality is available through the RCX 2.0 Beta SDK [LEG00].

2.2.1.1 User interface

The software that comes with the RIS provides a complete environment for programming Mindstorms robots. The core of the RIS software is an interactive visual flow charter shown in figure 2.6. Lego bricks represent actions (e. g. turning on a motor), sensors, and flow control (like conditional processing or loops). The whole user interface is designed for ease of use, including step by step tutorials with videos accompanying each step.

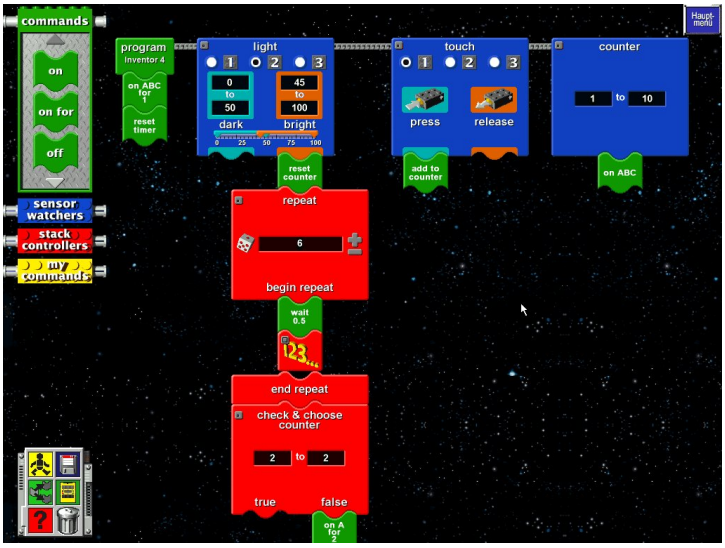


Figure 2.6: The Robotics Invention System programming environment

Sensor inputs can be processed with so called *sensor watchers*. Each type of watcher is responsible for one of the possible event sources listed in section 2.1.1.1 and has one or more connections points. The light sensor watcher, for example, has two connection points – one for dark and one for bright values. The commands under the dark connection point are executed whenever the sensor value enters a configurable ”dark” range. The commands under the light connection point are executed whenever the sensor value enters a configurable ”light” range.

The RIS software only provides a single counter that can just be incremented and reset to 0 again. The only way to react to the counter's value is a sensor watcher with a configurable range of values and a single connection point. Whenever the counter enters the specified range, the commands under the connection point are executed. The RIS software has no support for variables.

Only one of the RCX's timers is available in the RIS environment, and only with 100 ms resolution. All you can do with the timer is reset it and use it as an event source (represented by the timer watcher).

Commands that should be executed directly after pressing the Run button on the RCX – rather than waiting for an event – must be placed under the ”program” brick.

2.2.1.2 Generated programs

The programs generated from the flow chart consist of three kinds of tasks:

- The main task initializes timer and counter, the sensor inputs, and the motor outputs. Afterwards, it continuously monitors the events.
- The program task executes the commands placed under the ”program” brick. It is started by the main task after initialization is complete.

- One task per sensor watcher connection point. As soon as the corresponding event is triggered, the task is started. If it is still running when the next event arrives, it is restarted abruptly.

2.2.2 RoboLab

Like the software that comes with the RIS, *RoboLab* [LEGb] is a visual tool for creating programs in the form of flow diagrams. It is based on LabVIEW by National Instruments and available from Lego Dacta, the educational branch of Lego. RoboLab is much more powerful than the RIS software while still easy to use. Nevertheless, it still doesn't offer as much freedom and flexibility as the text based languages presented in the following sections.

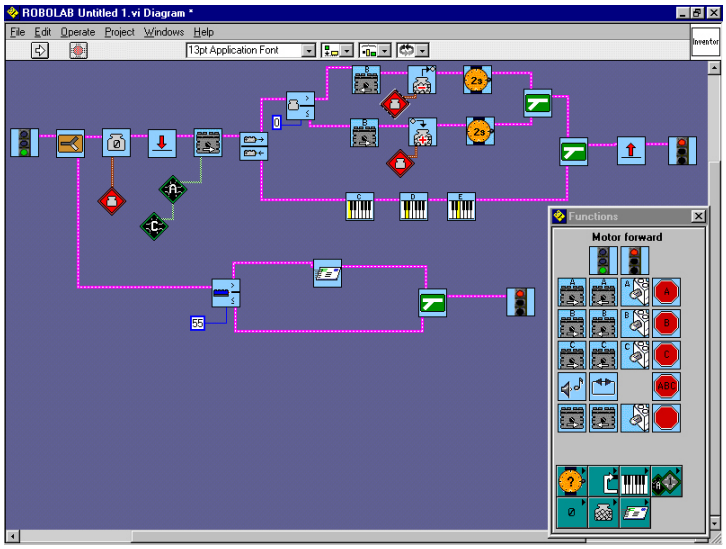


Figure 2.7: Screenshot from RoboLab
(Source: http://www.ceeo.tufts.edu/graphics/robolab.html)

2.2.3 LASM

The RCX 2.0 Beta SDK [LEG00] allows to program the RCX on an assembly language level. Each mnemonic directly corresponds to a firmware command. The suitability of LASM for writing programs is questionable. If you need to go to such a low level to achieve what you want, you should consider switching to a replacement firmware that works on another basis (see sections 2.1.1.1 and 4.2). LASM is, however, very useful for sending immediately executed low-level commands to the RCX without looking up the right bytecode and parameter format all the time. This can, for example, be used to quick-check the battery loading status or to read the global motor settings.

All necessary documentation can be found in the RCX 2.0 Beta SDK [LEG00].

2.2.4 The Spirit ActiveX control

The RIS 1.5 comes with an ActiveX control for programming the RCX from high level languages like C++. However, the necessary documentation [LEG99] must me downloaded seperately. Basically, the Spirit control wraps the firmware commands with methods. Each time you call such a method, the corresponding command is sent to the RCX. In addition to this ”direct mode”, the control also provides a ”download mode” in which several method calls are accumulated and then sent to the RCX as tasks and sub routines respectively.

With the Spirit control, you can create a complete programming environment for the RCX on your own, but I don’t recommend it for simply writing programs – since it’s just a wrapper, you basically use the LASM assembly language with its mixture of low and high level commands imposed by the firmware (see section 2.1.1.1). However, the control is useful for quickly building a ”control center” that displays status information, allows for downloading programs, etc. In fact, the RIS programming environment uses the Spirit control for communication with the RCX.

2.2.5 Lego Mindscript

The RIS 2.0 and parts of the Vision Command software (see section 2.2.7) use a script language called *Mindscript* for RCX programs. The most important features which make it preferable over LASM are expressions (e. g. `3*4/someVariable`) and macros. Macros are like functions in other programming languages and can have parameters (but no return value). They are either downloaded to the RCX as sub routines or expanded inline, whichever makes more sense (decided by the Mindscript compiler). Besides these advantages, Mindscript is also much more readable than LASM, although it’s not easy to get used to its syntax. In General, NQC (see next section) is the better alternative.

The RCX 2.0 Beta SDK [LEG00] contains a small application that allows writing and compiling own Mindscript programs. It also contains the Mindscript documentation as PDF and a Windows Help file describing the most important issues.

2.2.6 NQC

NQC stands for *Not Quite C* and is a programming environment developed by Dave Baum [Baua]. It supports a number of Lego programmable bricks, including the RCX with RIS 1.5 and 2.0 firmware.

NQC comes as a command line tool for compiling C like programs – hence the name – and downloading them to the RCX. It is also capable of reading the datalog (see section 2.1.1.1), downloading firmware, directly executing commands, and changing RCX settings like the range of the IR transceiver. Therefore, it can completely replace the RIS programming environment. NQC relies on the capabilities of the Lego firmware, so it cannot be used with firmware replacements. Currently, there is also no support for the USB tower of the RIS 2.0.

The language is easy to learn for anyone with a little C experience. The most important differences from regular C are:

Support for multitasking A task looks like a function without parameters and return value and can be started and stopped with the built-in `start` and `stop` commands.

Datatypes The only datatype is `int` with a width of 16 bits.

Functions Funtions look like C functions, except the return type must be `void` and parameters can be passed by reference using the C++ syntax; example: `void func(int& i)`. Functions are always inlined, so you should use them carefully.

Sub routines These are declared like a function, but using `sub` instead of `void`. The difference is that they are not inlined but only stored once in the RCX. Due to limitations of the Lego firmware, there can only be 8 sub routines in a program. Also, a sub routine cannot call another sub routine since only a single memory location is used to hold the return address. Unlike Mindscript, NQC doesn't support parameters for sub routines, so you have to use global variables as a workaround.[2]

Variables Due to limitations of the Lego firmware, NQC only supports 32 global and 16 local variables. However, the programmer does not have to think about this but can simply follow the scope rules of C. NQC automatically assigns global or local storage locations as needed - until it runs out of them of course.

Arrays Support for Arrays is very simplistic. For example, they cannot be initialized nor can they be arguments to functions. There are also limits regarding the use of operators on array elements.

Operators Besides the regular C operators, NQC supports `|=` to set a variable to the absolute value of an expression and `+-=` to set a variable to the sign of an expression (+1, 0, -1).

Resources and events The resource control described in section 2.1.1.1 is implemented by the `acquire` statement, while events can be monitored with the `monitor` statement. Both of them use `catch` clauses to execute code when ownership of a resource is lost or when an event is triggered respectively.

Preprocessor The NQC preprocessor is verly similar to its C counterpart. It supports the `#include` directive as well as conditional compilation. There are a few special `#pragma`s, e. g. for reserving certain storage locations so NQC will not use them for variables.

API NQC comes with a set of API functions for accessing sensors, motors, etc. Everything offered by the current firmware can be achieved through this API. For special purposes or future firmware releases, the `asm` statement allows direct specification of bytecodes (hex notation, not LASM).

[2] This is the only real disadvantage of NQC. In all other aspects, it is much more powerful than LASM or Mindscript.

Figure 2.8: The Vision Command programming environment

2.2.7 The Vision Command programming environment

2.2.7.1 Basics

The Vision Command (VC) software combines image recognition capabilities with an interface for building programs based on what is recognized in the image. Like in the RIS environment, the programs are represented by Lego bricks. There are bricks for sound and music generation, camera bricks for capturing still pictures or video sequences, flow control bricks like loops, and special bricks for interfacing with an RCX (see next section). However, you don't need an RCX to work with Vision Command.

The software is able to recognize three different image attributes:

Light Reacts to light of a configurable brightness.

Motion Reacts to motion of a configurable speed.

Color Reacts to a configurable color.

For more flexibility, these attributes can be detected and treated seperately in different regions of the camera image. These regions do not have to be rectangular, not even coherent! Up to 9 regions are combined into a so-called *pattern*. For example, figure 2.8 shows a pattern with 4 regions. Only one pattern can be active at a time.

Every region has its own stack of bricks that is executed whenever light, motion, or color is detected (configurable per region). What is not possible is to have a region react to more than one attribute, e. g. motion and color. It should be noted that the stacks are associated with region numbers, not with the actual regions. This means that if you change the pattern in a program, the stacks still remain the same, even if they don't make sense in the new context or the respective region doesn't exist anymore.

Vision Command comes with 20 predefined patterns. This set can easily be extended by own patterns as described on Michael Gasperi's web site about VC [Gas]. You can also find a special pattern there that allows detection of different colors in the same area of the camera image, which is impossible if you only use the standard VC patterns.

Besides command stacks for different regions, every VC program has a main stack that is continuously executed. By default, it only consists of an infinite loop with a single "wait and watch" command. This command just waits for some time and meanwhile checks if one of the regions is active (i. e. if motion or light is detected or the specified color is sensed). If so, the command stack for this region is executed. After the waiting period has expired, control is transferred to the next command (which is, by default, the infinite loop containing the "wait and watch" command itself). If you take away this brick from the main stack, image recognition is effectively disabled. By adding your own commands to the stack, you can, for example, let a melody play while waiting for something to happen.

2.2.7.2 RCX interface

Vision Command alone is quite flexible, but on its own it can just play sounds and capture still images and video sequences. It gets much more interesting if you combine it with an RCX and some motors. For this purpose, the VC software offers special bricks for controlling the RCX, especially the motors. An example for using this is a tracking mechanism: Mount the camera on a turntable and make it turn left if movement is detected on the left side of the image, make it turn right if movement is detected on the right side. This way, the camera automatically follows a moving object.

For communication with the RCX, VC uses the IR tower of the Robotics Invention System. On first look, you would expect VC to simply send directly executed commands to the RCX when a region becomes active. However, the developers at Lego have chosen a different approach: When a VC program is started, a small Mindscript program is compiled and downloaded to slot 5 of the RCX. It contains all the relevant bricks – i. e. excluding still image and video capture and other commands not intended for the RCX – as Mindscript commands or calls to Mindscript macros shipped with the VC software. The "wait and watch" command mentioned in the previous section is implemented by waiting for the specified period and meanwhile monitoring incoming IR messages. When a region becomes active, VC sends a 1-byte message with the number of the region to the RCX which then executes the corresponding stack of commands.

The main disadvantage of the VC/RCX communication architecture is that VC only provides very few RCX commands – basically just motor control and sound. Most importantly, you cannot query any sensors from a VC program. This is like giving the robot an eye but taking away its ears and nose ... But as always, there are people who find ways around such limitations. On his Vision Command web site [Gas], Michael Gasperi presents a way of making the camera function as a smart sensor. By manipulating one of the Mindscript macros used by VC, control is taken away from the VC program in slot 5 and transferred to an arbitrary other slot. The program there can then evaluate incoming messages on its own without being limited to the RCX bricks provided by Vision Command. Another advantage

is that you are not restricted to the visual programming environment anymore but can also use NQC.

2.2.8 The Logitech QuickCam SDK

The camera contained in the Vision Command set is a Logitech QuickCam model. Hence, the QuickCam SDK [Log] can be used to access it without the Vision Command software. The SDK comes with a COM object called *Video Portal* that enables saving pictures and videos to disk and gives easy access to the image data. This is great for implementing improved recognition routines: You can simply write your own application using the Video Portal. However, you then have to add RCX communication functionality on your own.

Besides the QuickCam SDK, every other software for accessing USB cameras works with the Lego cam too, including popular video capture tools. What I haven't tried is whether you can use another camera with the Vision Command software, which could greatly improve image quality.

2.3 Glossary

Datalog — A series of integer values collected by an *RCX* program. The datalog can be uploaded to a host computer for evaluation, usually for debugging purposes.

Firmware — The operating system of the *RCX*. The firmware resides in the RCX's RAM and uses low-level ROM routines for interfacing the hardware.

IR tower — An IR transceiver connected to a host computer. The IR tower enables communication between the host and a *P-brick*, e. g. for downloading programs to the brick.

LASM — An assembly language for writing *P-brick* programs.

Mindscript — A higher programming language for writing *P-brick* programs.

Not Quite C (NQC) — A higher programming language for writing *P-brick* programs. NQC was not developed by Lego but by Dave Baum [Baua].

P-brick — Short for programmable brick. At the time of writing, the available P-bricks are the *RCX*, CyberMaster, and Scout bricks.

Pattern — The combination of up to 9 *regions* in the *Vision Command* software.

RCX	A special lego brick with an integrated microcontroller. The RCX and other *P-bricks* are the heart of the Lego Mindstorms product line.
Region	A term used in the *Vision Command* software. A region is a part of the camera image in which light, motion, or color is detected. Each region can have its own stack of commands that are executed upon detection.
RoboLab	A visual programming environment for RCX bricks based on LabVIEW by National Instruments.
Robotics Invention System (RIS)	A Lego set containing an *RCX* brick, an *IR tower*, lots of Lego Technic parts, and software for programming autonomous robots.
Slot	A storage location for an *RCX* program. The RCX provides 5 program slots.
Vision Command (VC)	A camera set from Lego that includes image recognition software and can be combined with the *Robotics Invention System*.

3 The soccer project

3.1 Planning the environment

3.1.1 Prerequisites

As mentioned in the introduction, my main goal was to make two or more robots play soccer. This has been done with Lego Mindstorms before – most notably for the RoboCup Junior league (see [LP00] and [Rob]). However, the RoboCop setup requires a special kind of ball emitting infrared light and needs more parts per robot than contained in one RIS set. In contrast, I wanted to create an environment using only standard parts from the RIS and Vision Command sets. This way, the setup can easily be copied and used for own experiments without the need for additional hardware. Also, the techniques used can be applied to other problems too, while the RoboCup solution is fixed to soccer (which, of course, has its own advantages).

Figure 3.1 shows the RoboCup setup. You can see the gradient from black to white on the floor (explained in the next section) as well as the smart ball.

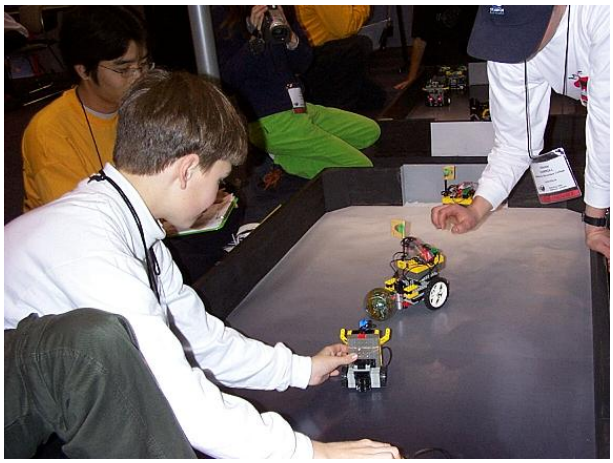


Figure 3.1: RoboCup Junior setup
(Source: http://www.demo.cs.brandeis.edu/rcj2001/rcj2k.html)

3.1.2 The environment used in this thesis

The main question regarding robot soccer is always: How do the robots find the ball? Since I didn't want to use any non-Lego hardware, the only sensible option was employing image recognition using the Vision Command camera. Since cable bound robots are not really a good idea for playing soccer, the camera must be mounted above the playfield, and the IR tower connected to the host computer must broadcast the current position of the ball to all robots. The resulting information flow is depicted in figure 3.2.

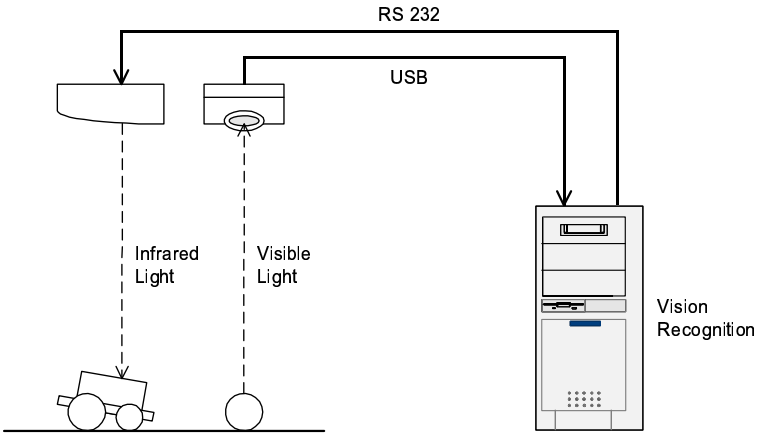


Figure 3.2: Ball recognition information flow

The playfield is devided into 8 quadratic regions, and the robots continuously receive messages with the number of the region currently containing the ball. Thus, they only roughly know where the ball is and are forced to search around a little. In order not to make it too difficult, the playfield is rather small with a length of 140 cm and a width of 70 cm. This yields a length and width of 35 cm per region, which should not be too large to find the ball.

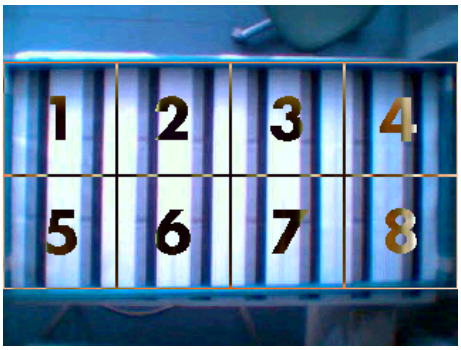


Figure 3.3: Ball recognition region scheme

Now that the robots know their destination, we're facing the next problem: How do they figure out where they are? In case of the RoboCup setup, a simple gradient from black to white is placed on the playfield, and one light sensor of each robot is pointing to the ground. This is enough for a player to decide whether it is moving towards the other team's goal, towards the own team's goal, or none of these. Together with the IR ball shouting out "here I am" all over the playfield, it's not too difficult to kick the ball into the right direction.

In my case, I needed a different approach. Since the robots only know the absolute coordinates of the ball, they also need some information about their own position to successfully kick it. While a gradient can quite reliably tell you where you're moving to, it's very hard to determine your actual position based on this. Besides, the gradient only works in one direction (up/down) and carries no information about the other (left/right). Nevertheless, I decided to stay with the general idea of a floor pattern and a down-pointing light sensor, but I looked for a different pattern. It also seemed a good idea to give the robots some abtract world model rather than completely relying on the floor sensor.

As for the ball, there are a few restrictions regarding its color. It must not be the same as the arena frame, and it should easily be distinguishable from the robots. Bright red is a good choice, but I ended up with a not so bright pink – the color of the best ball I've found regarding size (about the size of a tennis ball) and "bouncing capabilities" ...

In summary, the setup I planned consisted of an arena with a size of 140x70 cm, a floor pattern for orientation, an overhead camera, and four Mindstorms robots (two per team). And, of course, a ball. Pictures of the individual components can be found in the following sections. For programming the robots, I chose NQC (see also below).

3.1.3 Possible alternatives

What is most annoying about the environment described above is the need for an overhead camera. It would be much more elegant if the robots could find the ball themselves while still using a "normal" ball. A possible approach would be to employ the proximity sensing technique found in [Ove99], but there are two major difficulties: The technique only works over short distances, and the robots must differentiate between the ball, other robots, and the arena frame. There would be so many issues to solve that I don't think this solution can really be put into practice.

Another problem with the overhead camera is the inaccurate localization of the ball. Vision Command only allows for a maximum of 9 regions of which I use only 8. In my case, this yields a resolution of 35 cm for the ball's position. To improve this, one could make the playfield even smaller, but this is not really an option (see section 3.4). The other alternative would be dropping the Vision Command software and replacing it by something more specialized to the problem. This is further discussed in section 4.2.

If we really need an overhead camera, the question arises if we could also use it to tell the robots where they are? This would require markings on the robots and a special recognition software, for example presented in [RAWG01]. All in all, it doesn't seem to difficult. Again, that is prevented by Vision Command's limitation to 9 regions and color detection being constrained to these regions. Besides, telling the robots where they are – or even where their team mates and opponents are – would make them much less autonomous.

As for the robots' programs, Mindscript would be an alternative to NQC. However, Mindscript is not very readable, and NQC provides a lot more flexibility, e. g. regarding expressions. LASM is not really a choice if the program should stay reasonably understandable. Of

course one could also work with a non-Lego firmware, providing the option to use ordinary C or Java for programming. In order to keep the whole environment easy to copy and adept by others, I turned down this option (see also section 4.2).

3.2 Constructions

3.2.1 The arena

To prevent the ball and the robots from leaving the playfield, I built a simple wooden frame shown in figure 3.4. At first, I wanted to use Lego Duplo bricks to emphasize the ”Lego feeling”, but the amount of bricks needed was far too expensive. The frame is painted dark blue, so it cannot be mistaken for the ball. The space surrounded by it is almost exactly 140 cm long and 70 cm wide. A bigger arena would have been nice for more freedom of movement, but finding the ball would have been more difficult then (see section 3.1.2).

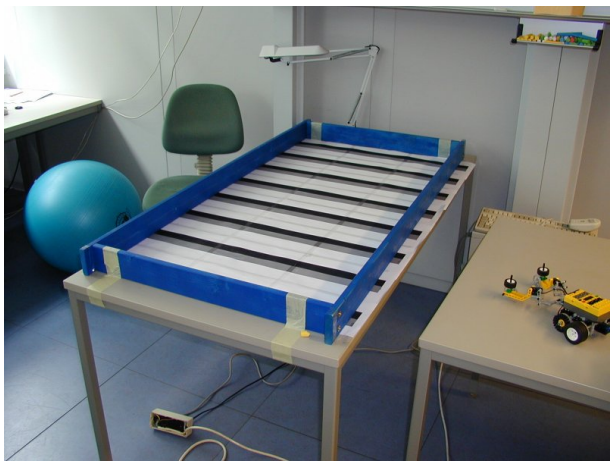


Figure 3.4: The soccer arena

3.2.2 The robots

First of all, I must admit that I’m not a gifted Lego builder. I find it fascinating to see Mindstorms robots move around, but constructing them is quite hard for me. For this reason, I simply started with the Roverbot described in the RIS contructopedia and equipped it with a down-pointing light sensor. Since the original roverbot moves quite slowly, I also replaced the motor and wheel gears in order to get a 1:1 gearing.

To detect touching the ball, the frame, or other robots, I added a 2-sensor bumper. Again, my starting point was the version from Lego’s contructopedia. By using two touch sensors, the robot can tell whether the left or right bumper arm was hit.

The main disadvantage of the standard bumper is its inflexibility. For example, a robot standing near the frame might not be able to turn around, just because a bumper arm slightly

touches the wall and blocks further movement. To solve this, I added some of the flexible tubes from the RIS (see figure 2.3 on page 9). The result is shown in figure 3.5.

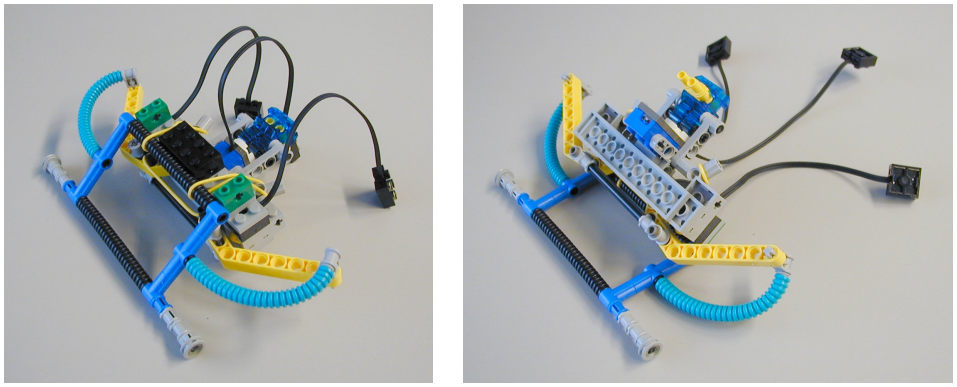


Figure 3.5: Bumper of the soccer robot

Figure 3.6 shows the complete soccer robot. As you can see, it takes up a lot of space, especially considering the small playfield. However, the bumpers turned out to be crucial for the robot's "sense of orientation", so it's difficult to make it considerably smaller.

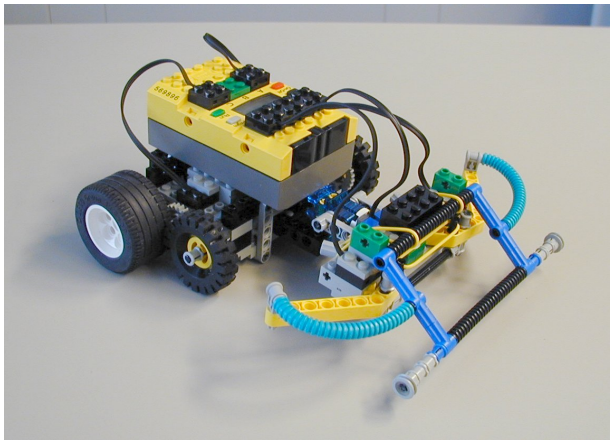


Figure 3.6: The soccer robot

3.2.3 The overhead camera

The construction for the overhead camera is very simple. A Lego frame holds the camera and is attached to a wooden bar. Additionally, the IR tower is mounted on one side of the camera in order to broadcast the current position of the ball. This turned out to be very convenient: New programs can be downloaded to the robots "on site", i. e. without lifting them from the playfield, carrying them to the host computer, programming them, and putting them back again. I think the idea of an overhead IR tower is also useful for other scenarios. As mentioned before, I was able to cover 1.6 meters without problems. In fact, it's not easy to **block** IR traffic when both the tower and the RCX are set to long range. One day, I had the tower on my desk and the robot under it, and I could still download programs (probably because of light scattering).

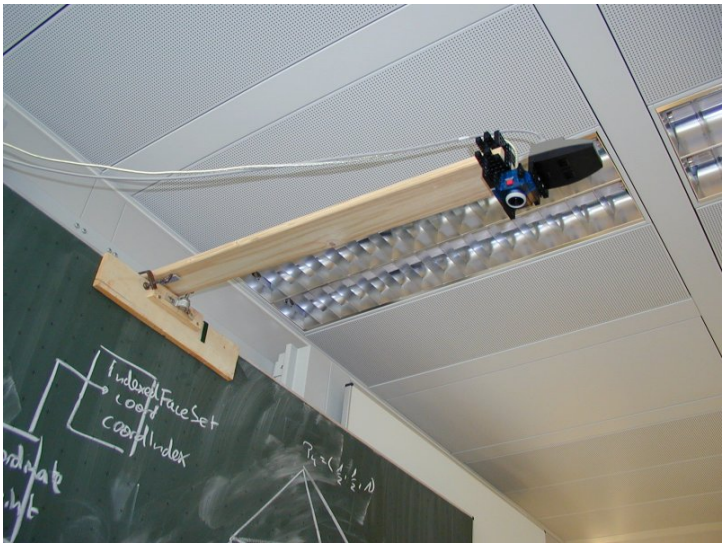


Figure 3.7: The overhead camera

3.3 Software

3.3.1 Where am I?

3.3.1.1 Floor patterns

As described in section 3.1.2, I planned to use a floor pattern for the robots' orientation. Unfortunately, this turned out to be quite a difficult task. My first approach was to simply code the different regions with shades of gray (figure 3.8). However, while allowing the robot to instantly determine its rough position, it holds no information about its orientation (i. e. where it's facing), so my next try were gradients with different lightness (figure 3.9). With this pattern, the robot can determine its rough orientation by checking if it gets darker or brighter. The region borders are a special case since they are marked by "contrast jumps" (at least in vertical[1] direction). The intention was to find out the region by analysing the magnitude of the jump. As you can see in figure 3.9, the start color of each gradient is black while the end is a shade of gray depending on the region. This makes the contrast jump different for each vertical transition of a region border.

As you may have guessed by now, this didn't work very well. I tried all other sorts of gradients, checkered patterns, and "bar codes" (figure 3.10). The main reason why they all failed was not that they were so bad (though most of them were probably nonsense) – the light sensor just wasn't accurate and fast enough.

First of all, the light sensor's default way of operation is to produce values between 0 and 100. During my experiments, I seldom got values below 25 and above 55. If you want to differentiate between 8 different shades, you get less than 4 values per shade! This wouldn't

[1] To save space, the figure is turned by 90 degrees, so vertical is only referring to the common assignment of x and y axis in this thesis. Regarding the figure, this should read "horizontal".

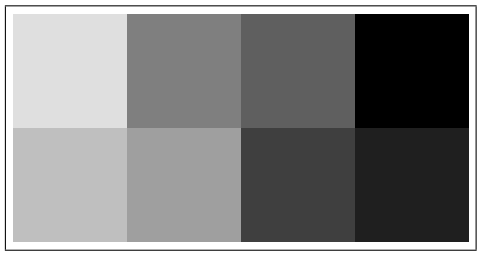


Figure 3.8: Floor pattern (first try)

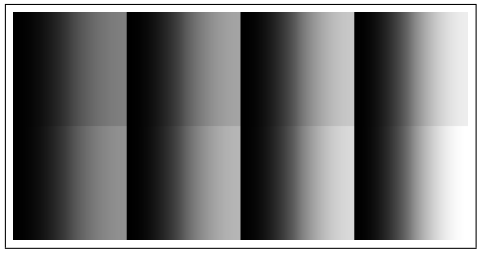


Figure 3.9: Floor pattern (second try)

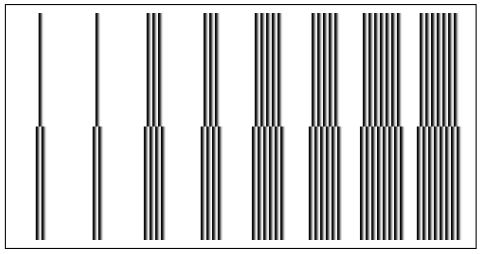


Figure 3.10: Floor pattern (third try)

Figure 3.11: Floor pattern (final version)

be a problem if the values were consistent. But instead, they change notably when overall lighting gets slightly darker or brighter. Having discovered this, I made two decisions:

- Use the light sensor in raw mode which produces values in the range [0, 1000] in theory and [600, 900] in practice.
- Concentrate on a few different shades and/or rapid changes, e. g. an abrupt transition from black to white.

But still there was one serious problem: All my patterns were much too fine grained. With a reasonably fast moving robot, I got about 10 samples per second or 1 sample per cm. This includes processing, because seeing may be believing, but you also have to evaluate the samples to determine the current position and orientation. However, even with simple processing I couldn't get much better. This was when I decided to rely much more on an internal world modal than I had originally intended.

In the end, I chose the very simple pattern shown in figure 3.11. Information about the horizontal position is drawn completely from the world model (see section 3.3.1.4). Vertically, the robot can tell its orientation as soon as it drives over one of the bars. This is simply achieved by looking whether a black/gray transition or a gray/black transition is encountered. Information about the current position can be collected by counting the bars during vertical movement. Anything else is left to the world model.

As you can see, the final pattern uses only 3 different shades. However, the robot is still sensitive to lighting changes, which is why I developed the adaptive algorithm shown in figure 3.12. It starts with initial values it expects for the different shades (`shades[0]` through `shades[2]`), but it also adapts these values with each sample it collects. For this purpose, a simple floating average is employed:

$$newShade = \frac{oldShade + newSample}{2}$$

In the beginning, I used a more complex formula, but multiplication and division are expensive operations.[2] Thus, in order to process as many samples per second as possible, I had to keep these operations to a minimum. In the final soccer program, the adjustment is even made conditional and only executed when there's a relevant difference between $oldShade$ and $newSample$.

If the current sample differs significantly from all three shades, if replaces one of them. This is done in a ring buffer fashion, so the first time `shades[0]` is replaced, the second time `shades[1]`, etc. What is this good for? At first, replacing the nearest shade, i. e. the one bearing the closest resemblance to the sample, seems to be the better choice. However, imagine the following situation: All initial values are between 750 (`shades[0]`) and 850 (`shades[2]`). Due to changes in lighting, the actual values lie between 500 and 600. In this case, the nearest shade is always `shades[0]`, so the others would never be touched.

[2] The Lego virtual machine doesn't provide bit shift functionality.

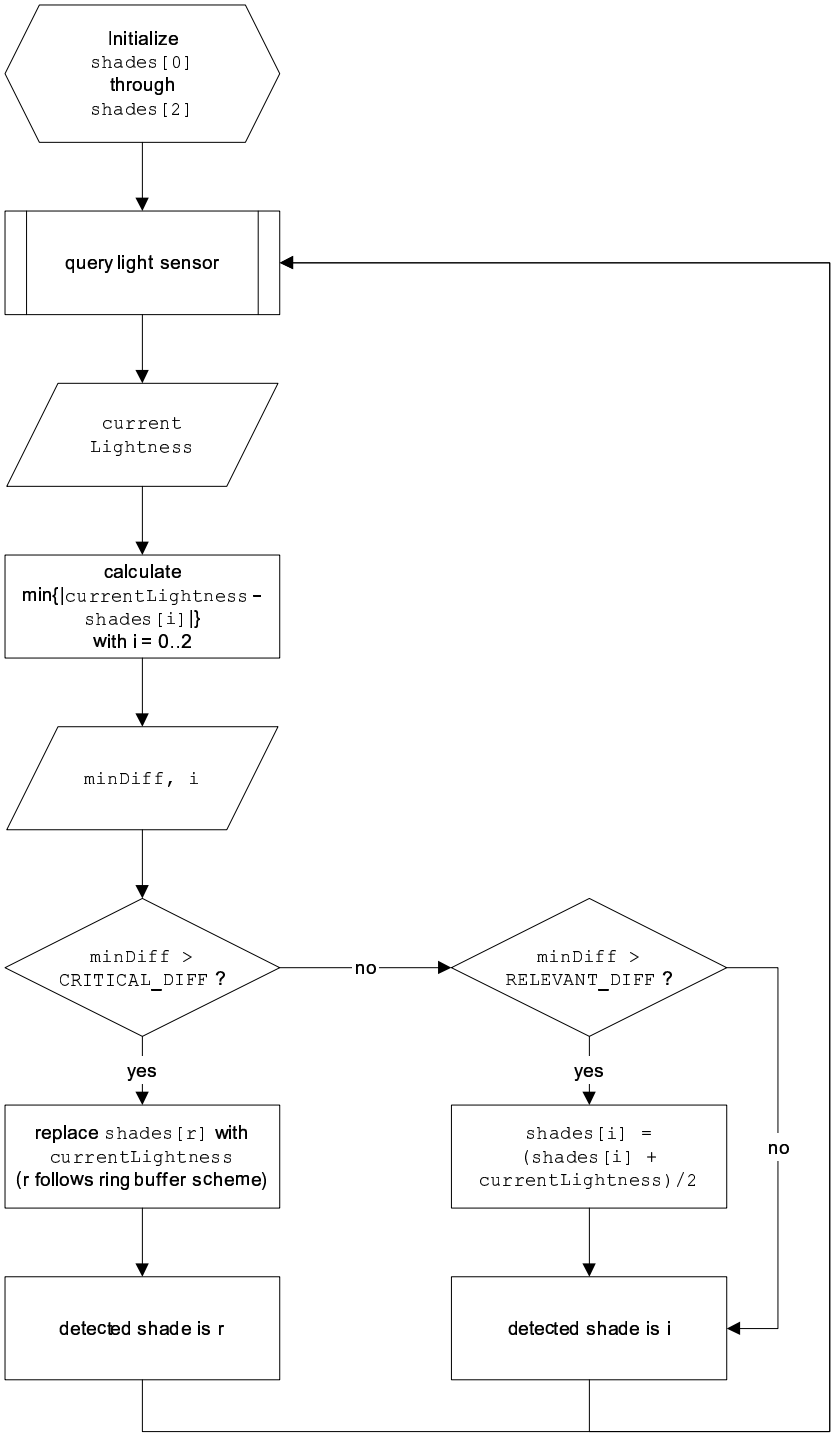


Figure 3.12: Adaptive detection of shades

Consequently, the same shade would be detected all the time. Of course, the ring buffer scheme in turn has the (minor) disadvantage that it often throws out the wrong shade, so the other have to be replaced afterwards too.

To test the algorithm, I made a robot move around on the floor pattern and produce a sound everytime a shade had to be overwritten. It turned out that this only happened in the beginning and very rarely afterwards. Even when the initial values were completely bogus, there were three beeps at the beginning (telling that all three shades were being changed), and only fast changes in lighting lead to another replacement afterwards.

The algorithm works fast enough for my chosen floor pattern and can easily be adapted to differentiate between more than 3 shades. However, I wouldn't recommend more than 4 or 5 for the sake of robustness. Also, it should run in a high priority task to avoid irregular sample timing.

3.3.1.2 Driving along a wall

As already mentioned, I decided to rely largely on an internal world model of the robots. This implies that a robot must first learn its position and orientation through external input in order to calibrate its internal model. It's no use to have a perfect imagination of the world if it has nothing to do with reality ...

To find an initial position and orientation, I found it extremely useful to let the robot drive along a wall. The idea is simple: Follow the wall until you bump into another one (signalled by both touch sensors pressed), then you know you are in a corner. By also examining the floor pattern (see figure 3.11), you can exactly tell your position and orientation – at least if you were driving in y direction. Otherwise you have to try again by turning a little and driving along the next wall. This way, you need at most two runs to determine where you are.

Now the interesting question is how to actually follow the border. First, I tried a variation of the popular line following algorithm:

- If you currently touch the wall, turn away from it.
- If you don't currently touch the wall, turn to it.

With some additions to avoid getting stuck, it kind of worked. But as you can imagine, this rather simple scheme leads to very choppy movement (which is still nicely put). Finally, after experimenting with different variations, I found the algorithm shown in figure 3.13[3] which performs very smoothly.

[3]The figure shows how to follow a wall to the right. To get the counterpart for a wall to the left, simply replace all occurences of "left" with "right" and vice versa.

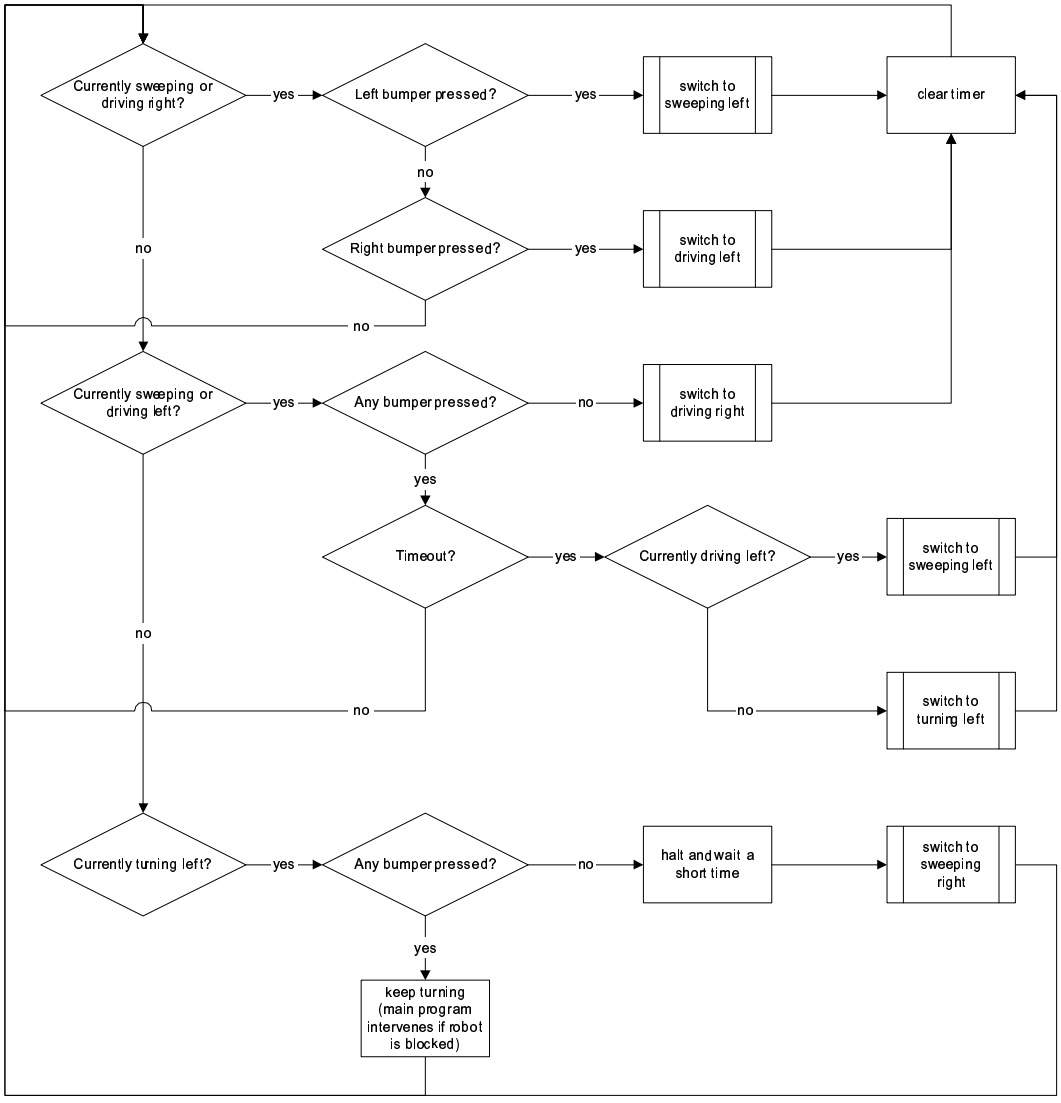


Figure 3.13: Wall following algorithm

This solution is based on three kinds of movement:

Driving In this mode, both left and right wheels turn in the same direction, but one side uses more power. Without surrounding walls, the resulting path would be a large circle.

Sweeping Similar to driving, but with one side turning very slow or not at all (”float” mode, see section 2.1.1.1). Without surrounding walls, the resulting path would be a smaller circle.

Turning Means turning on the spot by turning the left and right wheels in different directions.

To understand how the algorithm works, let’s first take a look at the example of driving along a wall to the right of the robot. The initial movement in this case is sweeping right (which is kind of arbitrary). Now let’s assume the robot touches a wall to the right, recognized through the right bumper. In this case, movement changes to driving left which slowly turns the robot away from the wall while still driving forward. As soon as the bumper is released, movement is changed to driving right. When the right bumber is pressed again, movement switches to driving left etc. The result is a smooth forward movement along the wall without any visible glitches.

Of course, the above example illustrates the ideal case. Often the robot is in an unfavorable starting position, e. g. in a corner (see figure 3.14). If the robot wants to follow the right wall in this example, the normal behaviour described above doesn’t work. Instead, an escalation scheme is employed that makes use of the spring-like behaviour of the flexible tubes attached to the bumpers:

1. Right and left bumpers are active. Since it wants to follow a wall to the right, the robot tries driving left.
2. After some time, the bumpers are still active. Out robot gets angry and tries sweeping left.
3. When it notices that sweeping doesn’t help either, the furious robot decides to turn left. Together with the spring effect of the right bumper, this is enough to ”break free”.
4. As soon as the right bumper no longer touches the wall, the now calm robot tries sweeping right – after all, it wants to follow the right wall.
5. Again, it gets stuck. However, this time the situation is a little better (the robot has turned a bit). It starts again at step 1 with the only difference that the left bumper is not active anymore. This continues until the robot has turned enough to successfully follow the standard scheme (driving left and right alternately).

Thanks to the flexible tubes, the same algorithm also works for the initial position shown in figure 3.15. In this example, the robot also wants to follow a wall to the right. At first, you might think it would be wise to turn right in this case, but the problem is that the wall often prevents that – the robot just gets stuck with its left rear wheel. Instead, the escalation

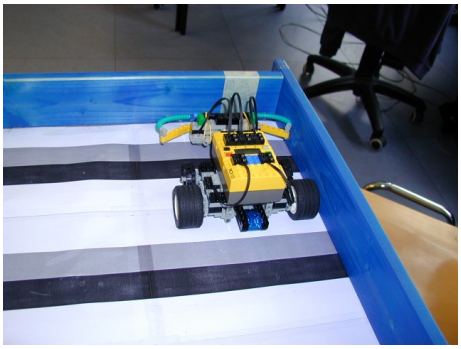


Figure 3.14: Robot in a corner

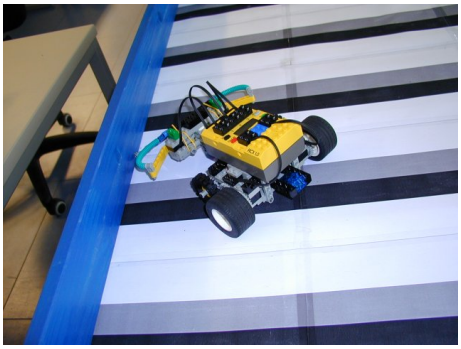


Figure 3.15: Robot at a wall

scheme is employed once more. As soon as the turning stage is reached, the spring effect leads to a slight turn after which sweeping right is tried again. At first this fails, so another little turn is made. This continues and lets the robot gradually turn while facing the wall. Finally, it has turned enough so the standard scheme can be applied.

Please note that the robot cannot tell which wall is touched by a bumper. Especially in a corner situation as shown in figure 3.14, the right bumper might either signal the wall to the right or in front of the robot. This is the reason for the escalation scheme. Without it, the robot could either choose to turn immediately (good for corners, but choppy movement otherwise) or not to turn at all (very bad for corners).

In case all the abovementioned measures fail to get the robot out of a tricky situation, the main program periodically checks whether the robot is stuck. If so, it tries to break free by executing random movements (see section 3.3.3.2).

A disadvantage of the algorithm is that it leaves its marks on the tubes (see figure 3.16). Partly, this is a result of my extensive test runs, but you should take it into account if you're planning to copy the setup or parts of it. After a while, of course, the movement becomes even smoother due to the ”polishing” ...

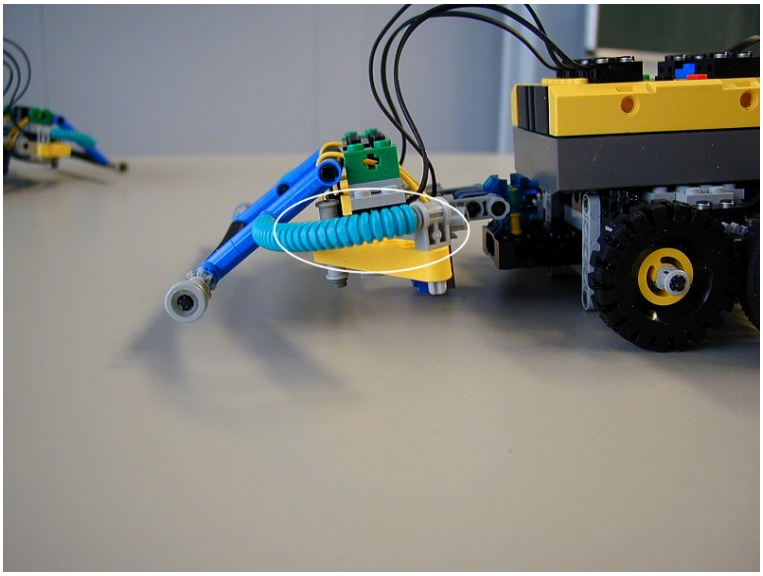


Figure 3.16: Unwanted ”polishing”

3.3.1.3 Aligning orthogonally to a wall

The wall following algorithm presented in the previous section is theoretically enough to calibrate the robot’s world model. However, it takes quite a while to finish. This wouldn’t be a problem if it was just needed in the beginning. But unfortunately, the world model gets inaccurate pretty soon and must be recalibrated to function properly. Therefore, I developed another building block for the robots’ orientation that allows facing a wall in an exact right angle. This way, bumping into a wall not only delivers positional information, but can also be used to achieve a defined orientation.

The aligment algorithm (figure 3.17) is simpler than the previous ones. If the left/right bumper is pressed, the robot turns left/right a bit and then drives forward again. This is repeated until both sensors are active which is interpreted as orthogonally facing the wall. Sometimes, this is not quite true, but most of the time this scheme works very well. An important aspect is that the robot drives forward for a while after turning, ignoring all sensor information. Without this measure, the robot easily gets trapped in an infinite loop as can be seen in the following scenario:

1. The robot drives across the playfield as suddenly the right bumper is pressed.
2. The robot decides that it’s time for recalibration, so it turns right a bit.
3. Driving forward presses the left sensor against the wall, so the robot turns left a bit.
4. Steps 2 and 3 continue endlessly ...

Even when the robot has reached an orientation that is sufficiently orthogonal to the wall, often one of the sensors reacts shortly before the other. Driving forward for some time and meanwhile ignoring sensor input gives the second sensor the chance to react too.

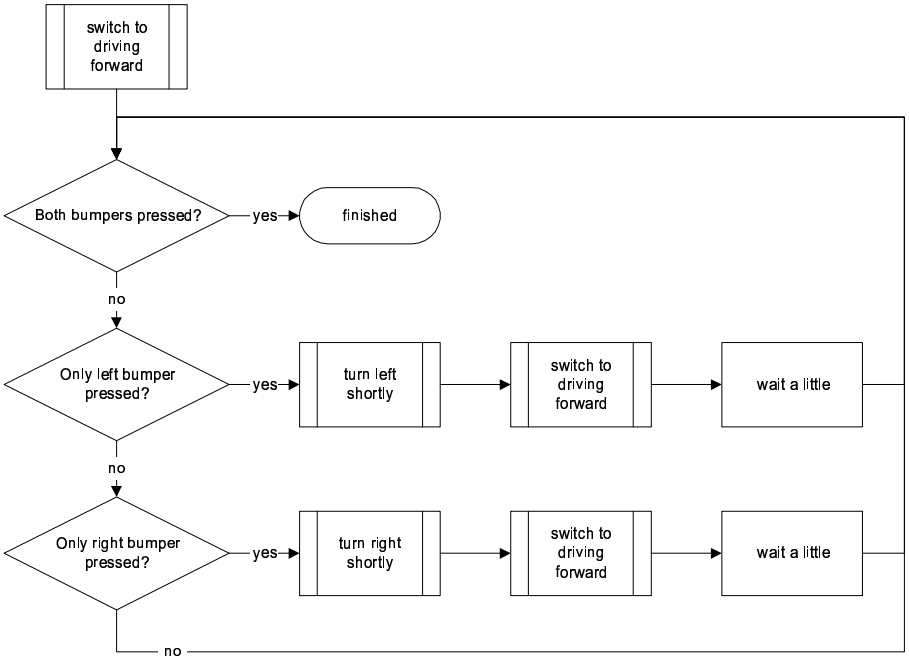


Figure 3.17: Orthogonal alignment algorithm

The alignment algorithm also works in very unfavorable situations like the one shown in figure 3.15. In this case, movement looks similar to the wall following algorithm, except turning stops when the robot directly faces the wall. In general, between 1 and 4 cycles of turning and driving forward usually suffice to reach the desired orientation.

3.3.1.4 The world model

By now, the robots are able to initially determine where they are and where they are facing to. Now all we need is a world model that updates this information without constantly receiving sensor input. The basic idea is to update the current position and orientation based on what **should** happen given the actions carried out (e. g. moving or turning). Once in a while, this information is updated by looking at sensor input or actively recalibrating using algorithm 3.3.1.2 or 3.3.1.3.

The core of the world model is an internal coordinate system. The first system I tried is very simple (see figure 3.18). One unit equals one mm. The placement of the origin and the directions of the axes resemble the way of representing screen or window coordinates in a graphical user interface. The current orientation is expressed using the constants `NORTH`, `EAST`, `SOUTH`, and `WEST`.

I restricted turning to a minimum of 90 degrees to keep operations on the world model simple – the Lego virtual machine is not good at math ... Also, turning cannot easily be controlled precisely, so you may end up with 80 or 110 instead of 90 degrees. The problem is that you can only influence the angle via the time used for turning. For example, turning left by 90 degress is achieved by starting to turn, waiting for some time, and then stopping again. Which time to choose must be determined through test runs with different values. Besides,

you might think measuring the degrees per second is enough to turn by an arbitrary angle, but accelaration and braking add non-linear effects. It's not easy to measure these exactly, so I determined the turning times seperatly for 90 and 180 degrees. Other values are not necessary for this model.

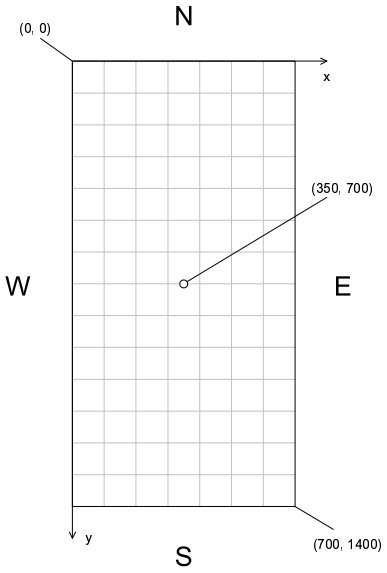


Figure 3.18: World model coordinate system (first try)

The coordinate system shown in figure 3.18 is easy to handle, but it has the disadvantage of being absolute. For example, when the robot moves forward either the current x or y coordinate must be increased or decreased, depending on the current orientation. The same is true for calibration by orthogonally facing a wall: Afterwards, either x or y must be set to either 0 or the maximum widht or height respectively. Althoug trying to avoid them by using advantagous numeric values for `NORTH`, `EAST`, `SOUTH`, and `WEST` together with modulus operations, I ended up with a lot of `if` statements and logical operators, e. g `if (direction == SOUTH && y < 1300)`, `if (direction%2 && x > 100)` etc. After a while, I got "out of memory" messages from NQC when trying to download the program, so I had to find a way to simplify my code.

The best solution I found was changing the coordinate system so it works robot-relative. This is depicted in figure 3.19. The robot always drives along the y axis, and it's always facing towards positive infinity. The origin is placed in the center of the playfield.

The new coordinate system makes it necessary to calculate a new position whenever the robot turns. For example, if we're at (100, 50) and turn right, the same (absolute) position lies at (-50, 100) afterwards. This effect could be avoided by placing the origin at the robot's current position, but then the coordinates of the walls would have to be updated with every move. Instead, with the origin in the center of the field, the wall in front is always placed at y = 700 or 1400 (depending on the current orientation which is remembered in an absolute fashion, see figure 3.19).

All in all, the new coordinate system made the program about 30 to 40 percent smaller. All complex operations are executed when turning which I could easily put in a sub routine that resides in memory only once.

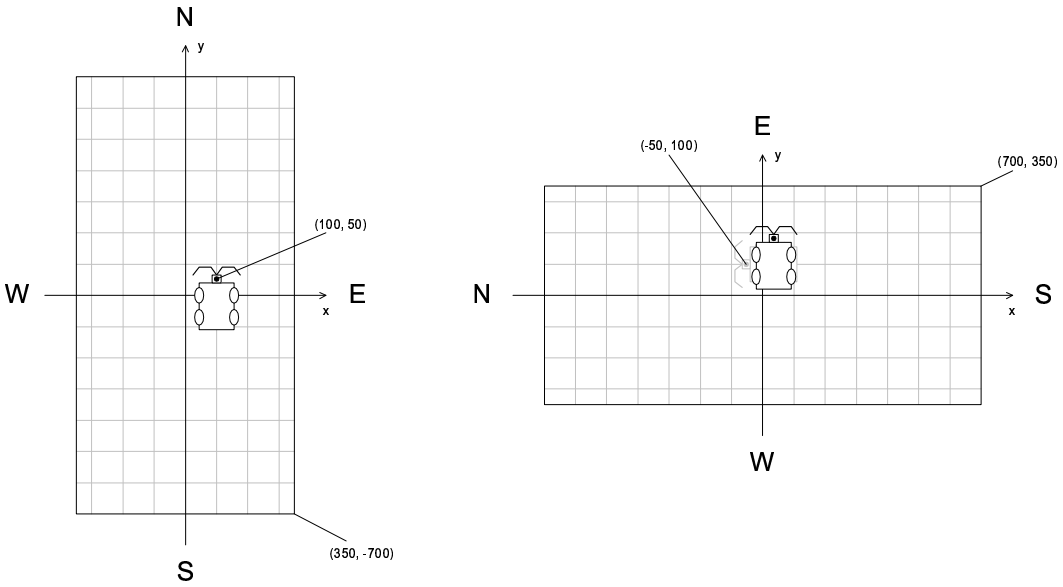


Figure 3.19: World model coordinate system (final version)

Naturally, there are not only advantages. Above all, the new system requires the ball's position to be translated into robot-relative coordinates upon receiving it as an IR message. Also, turning not only changes the robot's position but also the one of the ball. However, this is not much of a problem and clearly outweighed by the savings in code size.

There is one more issue regarding the world model that needs clarification. As you may have noticed, I always use the term "the robot's position" in the sense of an (inifinitely small) point with the components x and y. In reality of course, the robot has certain dimensions, so you have to choose one place that represents it as a whole. Since its position determines what the robot "sees", I decided for the light sensor. Thus, "the robot's position" really means "the light sensor's position".[4] The advantage is that adjusting the robot's position is quite easy this way. More specifically, if a bar of the floor pattern is encountered, you can simply take the bar's y coordinate (of course translated into the robot-local system) as the current y coordinate of the robot. If the robot was represented by a point other than the light sensor, you would need additional corrections.

3.3.1.5 Robot characteristics

In order to successfully update its internal world model, each robot has to know a little about itself – how wide it is, how fast it runs, etc. Tables 3.1 and 3.2 show these figures for the robots I used.

[4]Technically, the light sensor isn't a point either, but it's sufficiently small.

Attribute	*Value*	*Comment*
`MIN_DIST_LEFT_RIGHT`	95 mm	Distance from the light sensor to the left or right edge of the robot. Two times `MIN_DIST_LEFT_RIGHT` equals the width of the robot.
`MIN_DIST_TOP`	45 mm	Distance from the light sensor to the front end of the robot.
`MIN_DIST_BOTTOM`	140 mm	Distance from the light sensor to the back end of the robot.

Table 3.1: Static robot characteristics

Attribute	*Value*	*Comment*
`TIME_10_DEGREES`	40 ms	Time needed to turn by 10 degrees.
`TIME_90_DEGREES`	320 ms	Time needed to turn by 90 degrees.
`TIME_180_DEGREES`	590 ms	Time needed to turn by 180 degrees.
`DIST_PER_SEC_FULL`	500 mm	Distance covered when driving one second at full speed.
`TURN_DIST_NE_X`	55 mm	Distance along the x axis covered by turning 90 degrees.
`TURN_DIST_NE_Y`	70 mm	Distance along the y axis covered by turning 90 degrees.
`STEP_BACK_TIME_CORNER`	1000 ms	Time spent driving backwards to get out of a corner (see section 3.3.3.2).
`STEP_BACK_ANGLE_CORNER`	130 ms	Time spent turning to get parallel to the walls again after stepping back.
`STEP_BACK_DIST_X_CORNER`	60 mm	Distance along the x axis covered while stepping back.
`STEP_BACK_DIST_Y_CORNER`	345 mm	Distance along the y axis covered while stepping back.

Table 3.2: Dynamic robot characteristics

Regarding the dynamic characteristics, there are two special numbers that need some explanation: `TURN_DIST_NE_X`[5] and `TURN_DIST_NE_Y`. To understand these, you first have to consider that the robot's position is really the light sensor's position, but the robot doesn't turn around the light sensor. For this reason, the position changes with each turn. Even worse, x and y change differently. When the robot turns 360 degrees, the differences annul each other, but when it turns 180 degrees left, then 180 degress right, then again left and so on, the robot in fact slowly moves away from its starting point! This is reflected by `TURN_DIST_NE_X` and `TURN_DIST_NE_Y`. `TURN_DIST_NE_X` means the offset from the original x coordinate, `TURN_DIST_NE_Y` means the change in y direction. Please note that the coordinate system changes when turning (see 3.3.1.4), and that the offsets just mentioned are interpreted in the system that was in effect before the turn. Thus, executing a turn modifies the world model in the following way:

- `TURN_DIST_NE_X` is added (right turn) or subtracted (left turn) from the current x position.
- `TURN_DIST_NE_Y` is subtracted from the current y position.
- The current orientation is changed (easy due to appropriate numerical values for the `NORTH`, `EAST`, `SOUTH`, and `WEST` constants).
- Current x and y coordinates are changed to reflect the rotation of the coordinate system (see figure 3.19). For example, (100, 50) becomes (-50, 100) after turning right by 90 degrees.

An annoying fact is that all dynamic constants change significantly with decreasing battery power. To put it differently: the weaker the batteries, the greater the difference between model and reality. Since the current battery level can be read by the running program, the values could be adjusted accordingly. However, the results probably wouldn't be very accurate, and a lot of experiments would be necessary to determine the adjustment factors or offsets. Besides, the dynamic constants would have to be changed into variables to make them adjustable – and the Lego firmware doesn't provide many of these (see section 2.1.1.1).

3.3.2 Where is the ball?

3.3.2.1 Using Vision Command as a smart sensor

As described in section 2.2.7, the standard Vision Command software takes complete control over the RCX. In order to get it back, I used the method presented by Michael Gasperi in [Gas] which effectively turns the camera into a smart sensor. The RCX can run an arbitrary program and receives a message every time something is recognized in the image.

Given this possibility, finding the ball should be simple, I thought. The VC software can recognize color, so all I needed to do was to devide the playfield into regions (see figure 3.3

[5] "NE" stands for "north east" and means this distance is covered when turning from north to east.

on page 20) and tell VC the color of the ball. The software would then send IR messages about the region containing the ball, and the robots could start to play. There was just one tiny little problem with this (in every sense of the word ”tiny”) ...

The camera must be mounted high enough to overlook the whole playfield. However, a tennis ball from this height and with an image resolution of 320x240 as used by the Vision Command software has a diameter of only about 18 pixels. Of course, not all of these resemble the overall color closely, so we’re left with only a handful of correctly recognized pixels. Now you have to know that the Lego software requires a region coverage of at least 1 percent to say ”yes, I really see the specified color here”. In the pattern I use all of the 8 regions have a width and height of 80 pixels in order to take up as much space as possible. Otherwise, the camera would have to be mounted even higher. Unfortunately, 1 percent of 80x80 pixels yields 64 pixels, and this is more than the software ever recognized from the ball, no matter what color or camera settings I tried.

At that point, I had two alternatives. I could either write the recognition software all on my own, or I could try to tweak the Lego software. Since I figured it would be easier, I decided for the latter (more about that in section 4.2). The question was: How do I improve recognition of small areas of a distinct color without turning the Lego software inside out? The answer is given in sections 3.3.2.3 and 3.3.2.4, but first I want to describe some technical basics on which I have built.

3.3.2.2 Excursion: USB cameras and WDM streaming drivers

Windows WDM streaming architecture The Lego cam contained in the Vision Command set is a standard CCD camera connected to the host via the Universal Serial Bus (USB). It provides video and audio streams as well as still images. Under Microsoft Windows 98, 2000, and all newer versions, such streaming devices are usually accessed through the *Windows Driver Model (WDM) streaming architecture* (short WDM streaming). WDM as a whole is intended to simplify driver development and make a single binary device driver work on both Windows 9x and NT based systems. WDM streaming adds an additional layer to this model that supports fast processing of synchronous multimedia streams.

A lot of streaming functionality is identical for a great number of devices, so it makes no sense to include that in every device driver. Instead, Windows provides the so-called *stream class driver* including generic stream handling, while hardware vendors only write a *minidriver* for each specific device. The minidriver is responsible for actually communicating with the device and is called by the stream class driver when necessary.

Figure 3.20 shows the control and data flow from an application all the way down to the device. Please note that each transition from user to kernel mode is time consuming due to necessary validations and copying of data. The upper and lower filter drivers shown in the figure are covered in the next section.

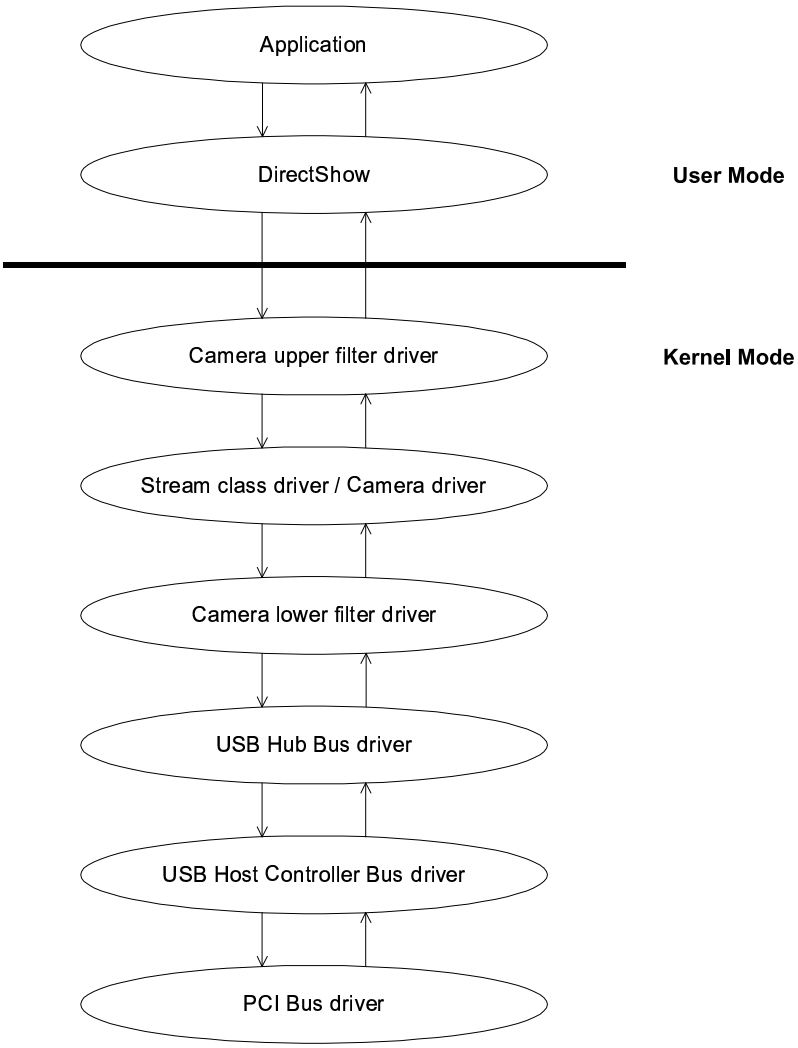


Figure 3.20: WDM streaming driver layers
(Source: [Mic99])

Each driver in the WDM streaming architecture is also called a *filter*[6]. Filters do not necessarily access a piece of hardware but may also just alter a video or audio stream. Each filter has one or more *pins* that allow connecting it to other filters. This way, a filter chain can be built that processes a stream in various ways. Since all the filters are kernel mode drivers, no transition to user mode takes place inbetween, and all filters can work on the original buffer without copying data. Thus, WDM streaming is clearly the fastest way to manipulate multimedia streams under Microsoft Windows.

One question is still open: Since an application usually doesn't "talk" directly to kernel mode drivers, how can it finally access the processed streams? This is achieved using the *Kernel streaming proxy module (KSProxy)*. KSProxy is a *DirectShow* filter that represents arbitrary kernel mode drivers. By wrapping the last driver in a kernel mode filter chain with a KSProxy instance, an application can access the stream like any other DirectShow stream.

More information about WDM streaming, including KSProxy, can be found in the Windows 98 Driver Development Kit (DDK) [Mic99]. DirectShow is described in detail in [Mic01].

Filter drivers The upper and lower filter drivers shown in figure 3.20 enable interception and manipulation of commands and data passed to and from a driver. Since this is done transparently and without the need to change any drivers or applications sitting on top, a filter driver is ideal for improving recognition of the ball. It can simply take the camera image, search for the ball, and amplify it so the Vision Command software "sees" it too. The difference between upper and lower filter drivers is evident from figure 3.20. Since we want to intervene after a stream frame has been received and processed by the camera driver but before it goes up to the application, we need an upper filter driver. Note that filter drivers can also be installed for class and bus drivers.

Communication with a driver occurs using *I/O request packets (IRPs)*. An IRP tells the driver to do something, e. g. initialize itself or go to sleep mode. Each IRP has a major and a minor function code. For example, the major code `IRP_MJ_FLUSH_BUFFERS` tells a driver to flush its buffers. All streaming operations are performed using `IRP_MJ_DEVICE_CONTROL`. Within such packets, an additional *IO control code (IOCTL)* specifies the desired operation, e. g. `IOCTL_KS_READ_STREAM` to capture a frame.

Filter drivers usually pass on all IRPs unchanged except those they want to filter. To suppress a request, the filter simply doesn't pass the IRP to the driver but indicates to the caller that the request has completed succussfully or failed. To manipulate the data resulting from a read request, the filter passes on the relevant IRPs and registers an *I/O completion routine* for each of them. The reason is that calling the original driver may only initiate a DMA transfer, so no data is available at first. The completion routine on the other hand is only called when the request was fulfilled, thus ensuring that all data has been received and can be changed. In case of write requests, the data is already available when the request is made, so it can be manipulated directly.

[6] Please note that this is not the same as a *filter driver*.

Finally, you should know something about *device objects*. A device object is a data structure that is maintained by the system for each instance of a device, device driver, and filter driver. Device objects come in three flavors:

Physical Device Objects (PDOs) are created by the ”parent” of a device, e. g. by the USB bus driver in case of a USB camera. Their main purpose is to handle power management and Plug and Play.

Functional Device Objects (FDOs) are created for every instance of a ”normal” driver (also called *function driver*).

Filter Device Objects are created for every instance of a filter driver.

Each device object carries information about the device, the associated driver, the current IRP, etc. Besides that, every device object points to a so-called *device extension* which is a global storage area usable by the driver. Since the device object is passed to most of a driver's functions, its the best place to store global information. The size of the device extension can be specified by the driver. For each device, the PDOs, FDOs, and filter device objects form a stack. Figure 3.21 shows how this stack looks like in case of the Lego cam.

More about filter drivers can be read in the Windows 98 Driver Development Kit (DDK) [Mic99]. The DDK also includes examples that can be used for own experiments.

3.3.2.3 Writing a filter driver

In order to amplify the ball, I wrote a filter driver for the Lego cam based on the USB filter example from the Windows 98 DDK [Mic99]. Since I've never written a WDM driver before – and since this thesis is not really about driver development ... –, the result is somewhat crude. A ”proper” filter should present itself as a stream source and in turn act as a client to the underlying driver. Instead, I intercept `IOCTL_KS_READ_STREAM` and manipulate the captured frames as needed. The disadvantage is that the filter driver cannot handle different data formats and thus only works with the Lego cam. For the same reason, it cannot be installed as a class filter handling arbitrary video input sources. However, it does what it's supposed to do.

For the sake of brevity, the filter driver is referred to as the *cam filter* in the following sections.

Installing a filter driver Filter drivers are installed using special registry entries. First, you have to locate the registry key for the driver that should be filtered. In the case of the Lego cam, the key is located here:

HKEY_LOCAL_MACHINE\Enum\USB\
VID_046D&PID_0850&MI_00\0USB&VID_046D&PID_0850&INST_0[7]

[7] Note that this is the key for video capture. There is another one for the built-in microphone of the camera.

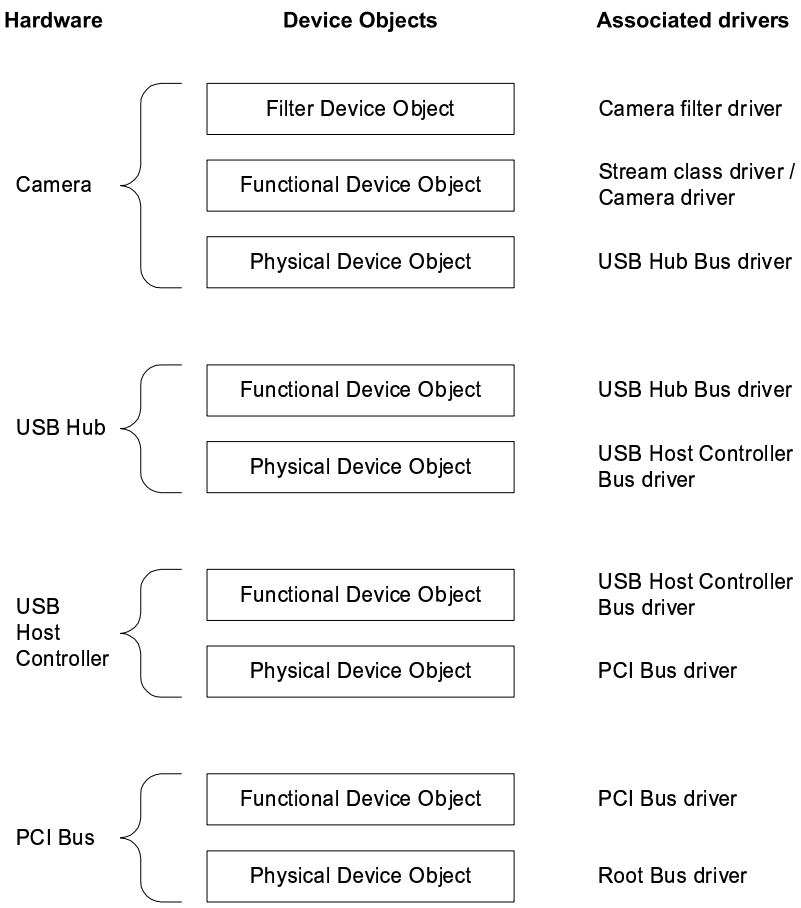


Figure 3.21: Device Object stack
(Source: [Mic99])

To specify upper and lower filter drivers, the values ”UpperFilters” and ”LowerFilters”, respectively, must be added to this key. There can be more than one filter driver of each type. Under Windows 2000 and up, the registry values are of type `REG_MULTI_SZ` with one string per filter driver containing the name of the driver file. Under Windows 98 and ME, these are `REG_BINARY` values consisting of a concatenation of all driver names, each of them zero-terminated, and a final zero byte at the end.

Reading and manipulating frames As mentioned, the cam filter is based on the USB filter example from the Windows 98 DDK [Mic99]. However, the example itself didn't work properly at first. Originally, the physical device object (see figure 3.21) was used to pass IRPs down the stack. This way, the functional device object and thereby the Lego cam driver were skipped, so no image showed up. After routing the call to the next lower device object instead, it worked.

The image data is manipulated by intercepting `IRP_MJ_DEVICE_CONTROL` with an associated `IOCTL_KS_READ_STREAM` (see section 3.3.2.2). Information about the image is contained in the `KSSTREAM_HEADER` structure passed with this IRP.

Before the cam filter does anything, it checks whether the `FrameExtent` field in the stream header equals 230400 (or 320 x 240 x 3). This way, it is ensured that there is valid buffer memory, so manipulation is safe. If the check fails, the Lego cam driver is called and no further action is taken. The buffer receiving the actual image is pointed to in the `Data` field of the stream header. In case of the Lego camera, the pixels are stored in 24 bit blue-green-red (BGR) format and organized from bottom to top, left to right.

After calling the driver – and provided `FrameExtent` had the expected value –, we could start changing the image. However, without waiting for completion of the data transfer, we don't know how many pixels have already been read. Therefore, the cam filter registers a completion routine to ensure the whole frame has been transmitted from the camera.

The completion routine is theoretically the right place to make changes to the image. Indeed this worked perfectly when using a simple video capture application. However, Vision Command and the underlying Logitech camera software made the cam filter crash. After a while, I found out the completion routine is called in another context than the IRP handler in this case. As a result, the pointer to the image data is invalid within the completion routine, and any access leads to a bluescreen. To resolve this, I changed the IRP handler to wait for an event signalled by the completion routine. Thus, the IRP handler can safely alter the image while still waiting until all pixels have been received.

Unfortunately, the abovementioned method did not always work. At times, only parts of the image were changed, as if there was no synchronization at all. It turned out that another IRP came in before its predecessor was completed, so the completion event waking up the IRP handler actually belonged to the **previous** request. To avoid this, I installed a counter that is incremented with each IRP and decremented with each call to the completion routine. Only when the counter reaches zero again, the event is signalled. Of course, this procedure could lead to slower performance, but I couldn't discover any such effects.

In summary, the technical frame of the cam filter performs as expected. Nevertheless, I would have implemented it differently with a little more WDM streaming knowlegde. A ”proper” filter driver should not just intercept communication between the camera driver and applications – this might lead to stability problems or loss of performance as pointed out above. Rather, it should present itself to the higher layers as a video stream source and in turn act as a client to the original driver.

3.3.2.4 Finding and amplifying the ball

The previous section showed the system-technical details of manipulating the Lego cam’s image before it arrives in the Vision Command programming environment. Now I’d like to explain how the ball can be found even under unfavorable circumstances such as non-uniform lighting.

Basic technique Finding the ball doesn’t seem to be a difficult task – just look for pixels of the right color, and that’s it. The first version of the cam filter exactly worked this way. Given the ball’s color as RGB values, it looked for similar colors in the image using a configurable tolerance. However, the results were not very impressive, mainly because of the following points:

- The Lego cam’s image is of poor quality, especially over a distance of 1.6 meters as needed in the soccer environment.
- It’s difficult to get good lighting for the arena. In my workplace, the ceiling lights interfered with the camera image by producing flickering horizontal stripes, so I turned them off. Since I didn’t want to use a huge flood light to compensate for this, some corners are always darker than the rest.
- The camera produces color fringes at edges with sharp contrast. This is a side effect of the CCD technology and the low resolution of the Lego cam. Unfortunately, some of these fringes contain pixels similar to the pink ball I used.

As for the non-ball pixels that are detected, I developed a simple solution that can at the same time be used to amplify the ball. The problem is to find a cluster of pixels with certain minimum dimensions and ignore smaller ones. This way, (hopefully) only the ball is recognized while fringes and noise are not considered.

Figure 3.22 shows how this is achieved. An array of dimensions 320 x 240 elements holds a ”weight” for each pixel. The higher the weight, the more likely it is that the pixel actually belongs to the ball. At the beginning, all weights are initialized to 0. For each ”ball candidate” pixel, the weight array is incremented by 1 in a circular area around it. For example, figure 3.22 shows three ball candidates (marked with small circles). The incremented areas are marked with R, C, and X respectively. A square containing one letter stands for a weight of one, a square with two letters for a weight of two, and a square with all three letters for

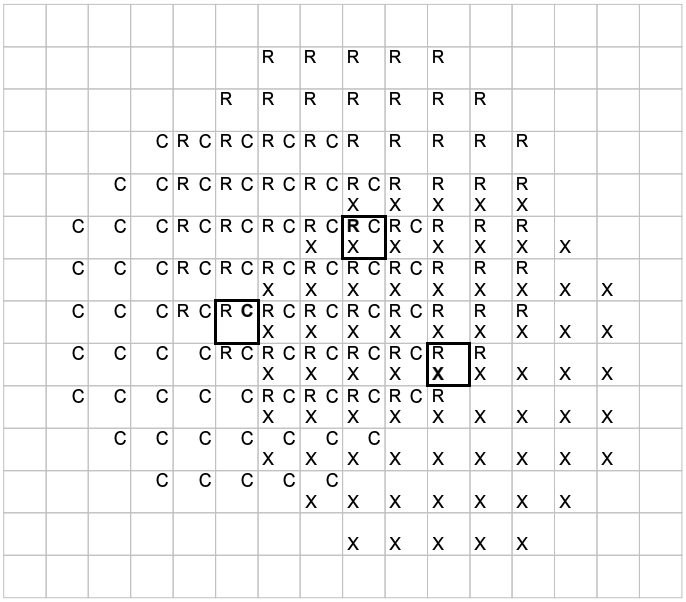


Figure 3.22: Example for using ”weights” to find the ball

a weight of three. At the end, all pixels with a weight above a certain threshold are colored bright green in the actual image, so the Vision Command software cannot miss it.

Besides finding its pixels, this method can also be used to enlarge the ball. The higher the radius of the ”increment circle”, the larger the amplified ball image. You just have to be careful to increase the weight threshold together with the radius to avoid false recognitions (it’s easier for the increment circles of disturbing pixels to overlap when the radius is greater).

Technically, the weight array resides in the cam filter’s device extension (see section 3.3.2.2), so it is allocated only once for each driver instance. The camera image itself is not copied but manipulated in place.

Color spaces One of the problems mentioned above is that there are lighter and darker areas in the image. However, the cam filter should recognize the ball no matter where it is. To achieve this, I had to change the color model from RGB to HSV (Hue - Saturation - Value, see [FvDFH96]). HSV is much better suited to the problem since it allows detection of a certain color (= hue) no matter how saturated or bright (= value) it is.

Implementing an RGB/HSV conversion is not a big thing - except if you cannot use floating point math. Due to some strange limitations, a WDM driver can only use the floating point (FP) unit if it performs initialization and saving/restoring of FP register values on its own. Of course I could have included the necessary assembler instructions after ”some” research, but I decided to employ a simple fixed point math solution instead with each number represented by 32 bits. 16 bits are used for the integer and 16 bits for the fractional part.

After successfully implementing the conversion to HSV, ball recognition became much better, but performance also went down notably. As a consequence, I started to ignore all pixels at an uneven x or y coordinate, effectively reducing the resolution to 160 x 120. This doesn’t

present a great loss of information since color is sent by the camera at this resolution anyway (YUV model with 4:2:0 downsampling), resulting in 2x2 blocks of recognized/unrecognized pixels. Before switching to HSV, this effect was not present since RGB distributes luminance and chrominance over all three components.

Customization To customize the filter, a set of registry values is used that must be placed under the key of the Lego cam driver (see section 3.3.2.3). The cam filter reads them everytime a frame is processed, so changes can be made on the fly. The following values are supported (all `DWORD`s):

`_color` The color of the ball in HSV format: 0xHHSSVV.

`_radius` The radius of the ”increment circles” as described above.

`_replacement` The color used for the enlarged ball in RGB format: 0xRRGGBB. This is what Vision Command gets to see.

`_showFireflies` If this value is different from 0, the ball is not amplified but all recognized pixels (i. e. having a color similar to the ball) are inverted. This way, the other values and the camera settings can easily be adjusted for best results. The name comes from dark pixels looking like fireflies when inverted (see figure 3.23).

`_tolerance` The tolerance regarding H, S, and V. For example, the value 0x1840ff means that devitations of up to 0x18 (H), 0x40 (S), and 0xff (V) from `_color` are tolerated when looking for ball pixels. The deviations are absolute, not relative. In the example, any value for V is accepted since a deviation of more than 0xff cannot happen, no matter what `_color` contains. Pixels for which H cannot be determined (shades of gray) are generally ignored.

`_verticalFilter` When different from 0, makes the cam filter ignore narrow vertical lines of recognized pixels. This measure helps reduce the effect of color fringes.

`_weight` The weight threshold for recognizing ball pixels.

A command line tool for installing the cam filter and modifying these values can be downloaded from [Juna].

Summary The camera filter driver enables Vision Command to recognize the ball. You can see the effect by looking at figure 3.23. The leftmost picture shows the original image, the one in the middle emphasizes the pixels recognized, and the rightmost is what Vision Command gets to see. On the PII/350 machine I used, the filter introduces an additional latency of less than half a second. This means that motion is still shown fluidly, just a little delayed. Given the needs of the robots, this is perfectly tolerable. I suppose my crude WDM driver code is the reason why there is a visible latency at all. Also, it only occurs in Vision Command and the Lego QuickCam software. A simple video capture application shows the filtered stream almost without delay.

There are still two important possible improvements to the cam filter. First, the weight threshold could be determined automatically by the filter. The simplest way of doing this would be to take the maximum weight encountered and use it as the threshold (provided it exceeds a certain value). The advantage is that lengthy experiments for finding the best `_weight` could be avoided. This isn't already implemented simply because I thought of it only a few days before my deadline ...

The second improvement would be to let the filter adept itself to different lighting situations like the floor pattern algorithm shown in section 3.3.1.1. Also, the filter could try to determine the ball's color itself, maybe by trying several possibilities and looking at the result until exactly one round object of the expected size is found.

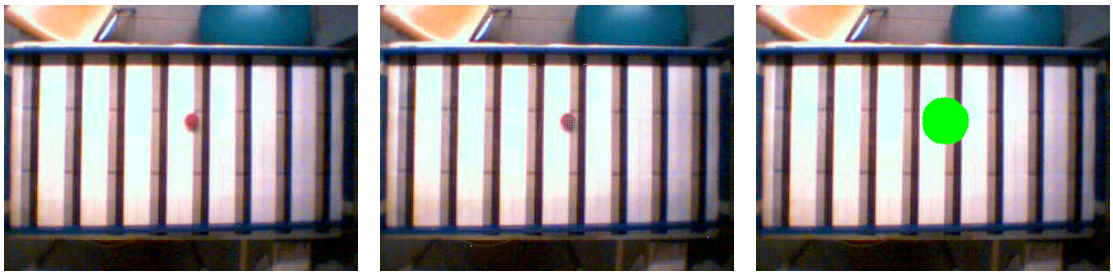


Figure 3.23: The cam filter at work

3.3.2.5 Broadcasting to several robots

Besides limited recognition capabilities, using Vision Command as a ball finder has another big hitch. When sending out an IR message, VC checks whether there is a response. If not, it displays an error message and stops the recognition. With only one robot on the field, everything is fine. However, when two or more RCX bricks are present, interference leads to a garbled response, and VC stops processing. Even worse, VC sends out "keep alive" commands periodically, so turning on more than one RCX in reach of the IR tower is already fatal, even when no recognition messages are sent.

To solve this issue, I looked at the path of communication from Vision Command to the RCX. To translate Mindscript, download programs, and send direct commands to the RCX, Vision Command uses a DLL named `VPBrick.dll`. I figured that the easiest way of supressing error messages would be to fool Vision Command into thinking that every message transmission is successful. To achieve this, I tried to insert a proxy DLL as as shown in figure 3.24. Its purpose is to route all calls to the original DLL and only intervene under certain circumstances.

I already had some experience with DLL proxies before (see [Junb]), but the problem with `VPBrick.dll` was that all exported symbols are "decorated", i. e. their parameters are coded into the symbol name like this:

```
?vpbExec@@YAJJPAVReaderWriter@@PAJ0P6GHJPAXJ0@Z2@Z
```

After some research, I found out that one can decipher this into the real parameters using the Windows API function `UnDecorateSymbolName`:

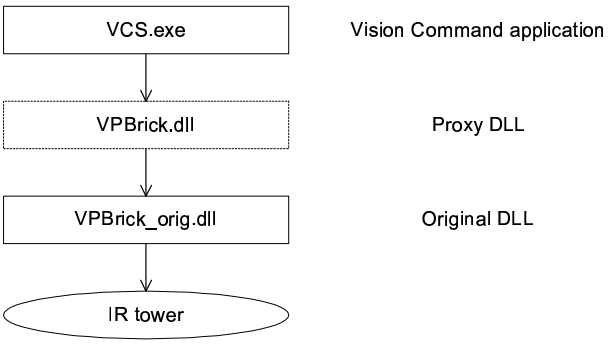


Figure 3.24: Vision Command RCX communication

```
long __cdecl vpbExec(long,class ReaderWriter *,long *,
  class ReaderWriter *,
  int(__stdcall*)(long,void *,long,class ReaderWriter *), void *)
```

With the help of `UnDecorateSymbolName`, I could build a proxy DLL with exactly the same functions as the original. At first, every proxy function was written in a way that just its original counterpart was called. After successfully testing this, I made the proxy write a log containing which functions where called with which parameters and return value at what time. Consequently, I was able to figure out that the `vpbExec` function is responsible for sending regular RCX messages as well as keep alive commands. After looking at the return values in case of success and failure, I found out how to modify `vpbExec` in the proxy DLL to fool VC into thinking that sending a message is always successful.

With the proxy in place, Vision Command can broadcast IR messages to several RCXs without stopping because of a garbled response. Up to now, I could not detect any negative effects of the manipulation. However, problems may arise when increasing the number of involved RCXs (yet to be tested). The only point you have to consider is that VC downloads a small program to the RCX when a recognition program is started. Since there is no way of simultaniously downloading a program to more than one RCX, you have to stick to the following sequence:

1. Turn on one RCX only and start the Vision Command program.
2. Wait until VC finishes downloading the generated program to the RCX.
3. Only now turn on the other robots.

If you want to use the proxy DLL for own experiments, you can download it from [Juna]. To install it, simply rename the file `VPBrick.dll` in the Vision Command folder to `VPBrick_orig.dll` and copy the proxy DLL to this folder. Important: Use exactly the name `VPBrick_orig.dll` for the original DLL – else the proxy won't find it. Also, be careful not to accidentally overwrite the original DLL, or you'll have to reinstall the Vision Command software.

3.3.3 The main program

3.3.3.1 Strategy

Having spent so much time on solving technical problems, I could only implement a simple but nevertheless effective playing strategy. The main idea is to drive the ball towards the opponent's goal while avoiding own goals. To achieve this, the robot employs an "always get behind the ball" strategy. Additionally, it tries to calibrate its world model on every favorable opportunity. The resulting behaviour can be seen in figure 3.25. This example shows the route taken by the robot to get behind the ball. When driving towards a wall, the robot always uses the opportunity to align orthogonally to it, hence losing some time but retaining a more accurate world model. Please note that the dotted lines in the figure indicate backwards movement.

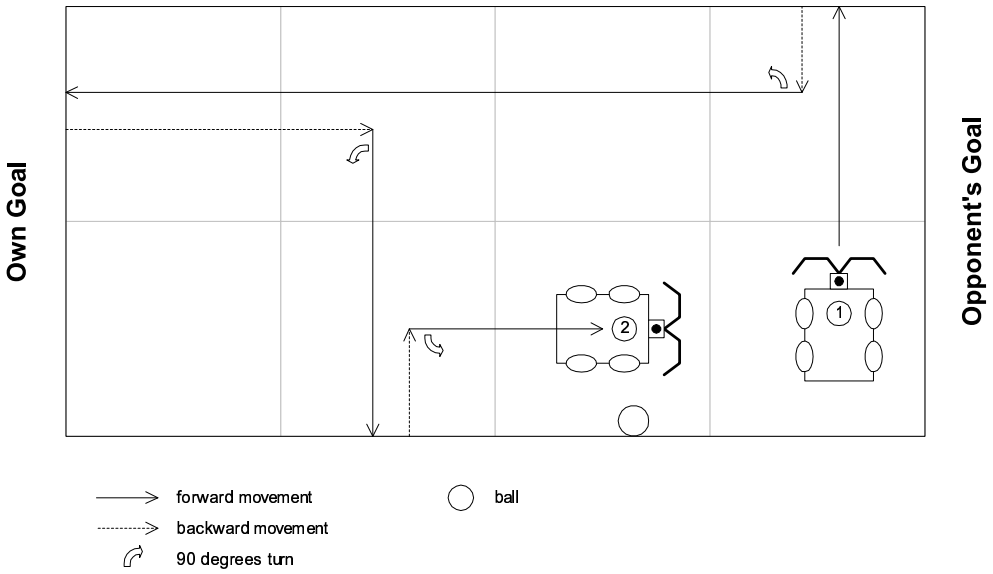


Figure 3.25: Playing strategy

As you could read in section 3.1.2, the robot just knows the ball's position by region, not exactly (figure 3.25 indicates the region borders in gray). To compensate for this, the robot checks whether it has failed to hit the ball by comparing its current position to the last one. If both are identical, the ball hasn't moved (at least not much), so it must be somewhere else in the region. First, the robot drives backwards, turns a bit to the middle and drives forward again. Both driving backward and forward are timed in a way that a distance equal to a region's width is covered. If this fails too, the robot goes back again and turns a little towards the wall. Now it doesn't simply drive forward, but rather goes into the mode of finding an initial position using the wall following algorithm (see section 3.3.1.2). This has two advantages: First, the ball often comes to lie close to the border or even in a corner. The only reliable way I found to get it out again was using the wall following scheme. Second, its sometimes necessary to repeat the inital positioning – just aligning to every wall that is encountered cannot correct all errors, e. g. not being able to make a turn without noticing it. The latter can make the world model wrong by 90 degrees, leading to a very confused style of play ...

3.3.3.2 Utilities and gimmicks

Getting confused It's very unusual, but getting confused can indeed be very helpful. How does this work? The robot increments a counter everytime a bumper is pressed unexpectedly. For example, if the robot drives towards a wall and expects to need two seconds for this, but after half a second a bumper is pressed, this counts as "unexpected". If the number of unexpected inputs between two calibration events (e. g. aligning at a wall) exceeds a certain limit, there is probably something wrong with the world model, and the robot reinitializes itself completely. Of course unexpected contacts also happen when bumping into another robot. This is why there is a counter instead of getting confused instantly.

Unblocking More often than you think, a soccer playing robot experiences a simple yet annoying problem - getting stuck. I've seen quite a variety of such situations, and the following list is far from complete:

- A robot has lost orientation and desperately drives against the same wall again and again and again.
- A robot "climbs up" the ball (see figure 3.26) and can't manage to climb down again.
- With the help of the ball, a robot manages to climb up the wooden frame (see figure 3.26).
- Two (or more) robots crash and try in vain to drive where they want to.
- Two robots get entagled with their bumpers.

Some of these situations could be detected using a rotation sensor, others not. For this reason – and also because I wanted to stick with one Robotics Invention System per robot – I decided not to use it. Instead, two techniques are employed. First, it is checked whether the light sensor input doesn't change notably over some time. This is an almost certain sign of being stuck. Second, a simple timeout guards against weird situations where the robot is still moving but cannot really get anywhere – which happens surprisingly often.

After realizing that it is stuck, the robot uses a simple unblocking technique: It selects a random movement like turning left or right or driving forward. After executing it for some time, the next one is chosen. This is repeated 6 times, hoping that the blocking situation is resolved afterwards. In case of failure, the timeout or the light sensor check triggers another try after a while. Usually, the problem is solved after 1 to 3 unblocking rounds. Of course, the robot completely loses orientation in the process, so it always tries to reach an initial position again after unblocking (see section 3.3.1.2).

Goals It always adds to the joy of watching autonomous robots if they show "emotions" suited to the current situation. For this reason, I wanted them to react to goals by shaking their head (own goal) or driving around joyfully while making music (successfully scored).

Of course, the robots must recognize goals to do this, which is not a trivial problem. You could use Vision Command's region scheme for this by introducing additional goal regions. However, this would reduce the number of regions available for the playfield, thus making it more difficult to find the ball. The next possibility would be to place another RCX close to the playfield and put bumpers with touch sensors into the goals. The additional RCX could then transmit goals to the robots as IR messages. However, in order to keep it simple, I decided for yet another alternative: The two leftmost and rightmost regions (see figure 3.25) are the goals. This way, it is very simple for the robots to recognize a goal and react accordingly. Also, it is sometimes the only way to tell which robot plays in which direction ...

Stepping back A disadvantage of the initial positioning described in section 3.3.1.2 is that the robot stands in a corner when it has found out where it is. In this position, it is impossible to turn or to drive forward – the only working movement is driving backwards. To make things easier for the strategy part of the program, I developed a "step back" routine that drives out of a corner backwards, but with a little drift towards the middle. However, after the robot is far enough from the wall, it is no longer aligned with the walls exactly, i. e. a non-rectangular angle is involved (caused by the drift). To correct this, the robot turns back a little. Finally, the world model is updated to reflect the new position. The strategy part can now safely make turns and be sure that at least the walls do not hinder the robot.

Fair play In order to facilitate a fair match, you have to avoid situations in which a robot completely blocks access to the ball. Therefore I force my robots to random movements regularly, even when the ball is in the opponent's goal, so the game always continues without human intervention. If the robots were more intelligent, I would have needed additional measures, like preventing intentional crashes etc. The way they play now, that's not necessary.

3.4 Put to the test

After putting all the pieces together, the robots could finally test their abilities in the arena. To verify the general strategy, I started with a single robot. Much to my surprise, it went really well. You could see clearly what the robot was trying to do, and it managed to get the ball out of all difficult situations. The only problem was that the ball often bounced back from the frame after scoring a goal and went straight to the robots own goal ... Nevertheless, the player always managed to keep the ball in the opponent's goal after a while, so you had to take it out by hand to keep the game running.

Next, two robots were sent into the arena and left to their own devices. Now it turned out that their dependency on the frame to keep their orientation is very hindering. Very often, one robot blocks the way to a wall the other is about to reach. As a result either the world model gets screwed or the two robots get entangled. However, as soon as a robot manages to establish a proper world model, it plays quite well and visibly enjoys its success (see section 3.3.3.2). Also, all blocking situations are resolved by the robots themselves.

It's hard to describe here how funny it is to watch the little fellows play, especially the situations when one places a nice shot and the other can only shake its head ... To give you an impression, some playing scenes are shown in figures 3.26 and 3.27[8].

It should be noted that human interference was hardly ever necessary during the test matches. With a bit of patience, you can let the robots solve their "conflicts" all on their own. The only situation that requires intervention is when a wheel falls off or a similar "accident" occurs. Regarding the ball, I have never seen a situation where the ball went somewhere and none of the robots could get it out again.

In the beginning, I planned 2 on 2 matches, maybe with one designated goalie per team. However, the arena turned out to be far too small for that. On the other hand, making it larger is bad for ball finding (see sections 3.1.2 and 4.2). Thus, I had to stick with one-"man" teams. What I also didn't test is using different strategies for the opponents. I was alreay happy to find **one** approach that worked reasonably well **and** fit into the RCX's memory ...

[8]These are actually faked since it's not easy to get the shots you want in the middle of the action.

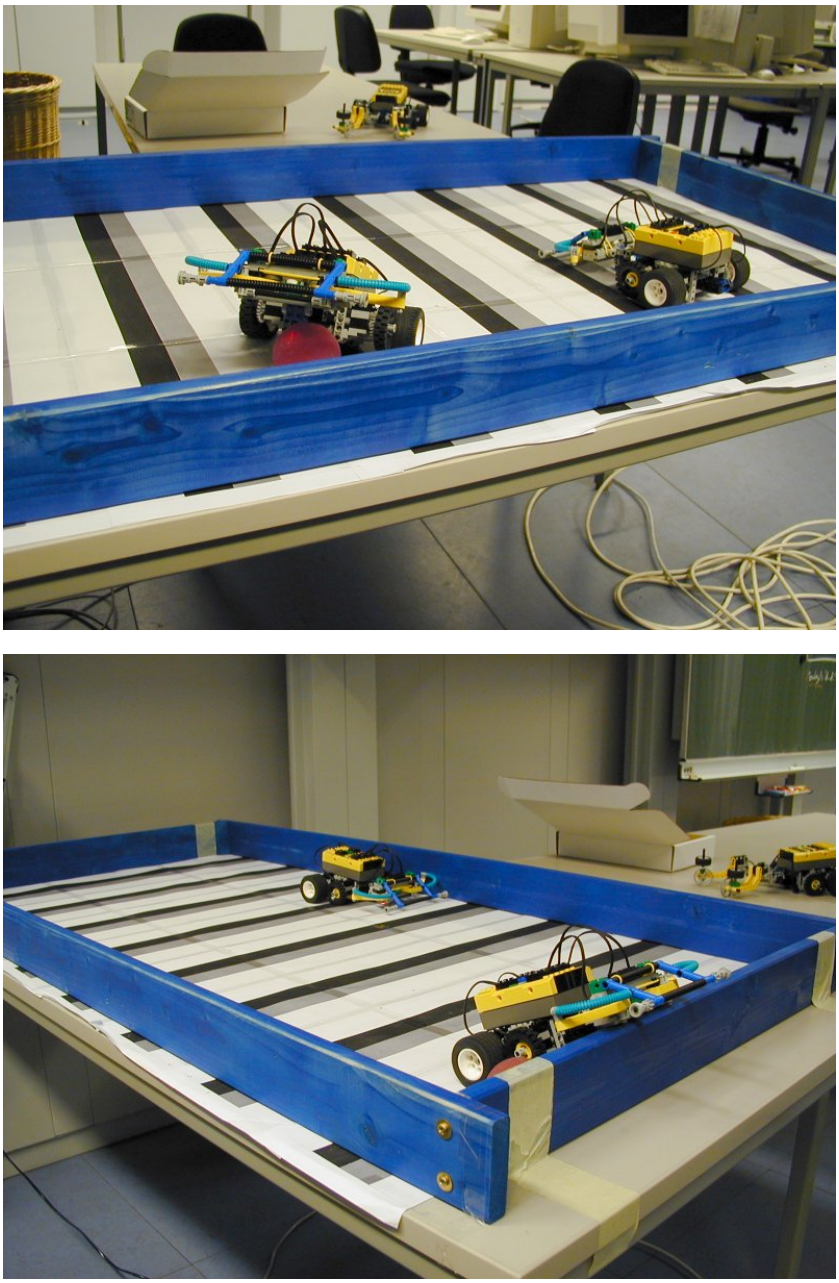


Figure 3.26: Master climbers

Figure 3.27: Go go go ...

4 Conclusions

4.1 Useful results

This thesis shows that it's possible to make Lego Mindstorms robots play soccer without any special hardware – ordinary RIS sets and a Vision Command camera are all that is necessary. What I did not achieve is realizing other ways of Mindstorms robots working or playing together since the soccer project took much too long. However, there are some achievements that I think are useful in other Mindstorms projects too:

- The proxy DLL (section 3.3.2.5) is useful for every environment in which Vision Command must broadcast to several robots.
- The camera filter (section 3.3.2.4) significantly improves Vision Command's recognition capabilities when it comes to small colored objects.
- The alignment and wall following algorithms (sections 3.3.1.2 and 3.3.1.3) can be helpful in any environment involving walls or other obstacles.
- The adaptive light sensor algorithm (section 3.3.1.1) can be employed in all situations where patterns on the floor must be recognized and lighting isn't constant or uniform.

All of these are freely available at [Juna], including source code. An excerpt of the NQC source is also shown in appendix A. If you have any questions or comments, please contact me via eMail (mailto:juan0014@fh-karlsruhe.de).

Besides getting familiar with Lego Mindstorms, working at the soccer project also taught me that it's often much easier to solve something in the form of hardware (e. g. "intelligent balls") rather than software. Instead of improving playing strategy, I spent weeks just to let the robots roughly find the ball.

Nevertheless, software has one big advantage: You can easily make it available to a wide audience via the World Wide Web or FTP. Everyone with an internet connection and a toy dealer in range can copy and improve the setup. If someone finds a great new strategy, he or she can send it to you, and you can test it against your own. If someone else constructs a better robot, you can build it and equip it with your own software. If something doesn't work, you can always find help on the Net. And thanks to Lego Mindstorms, you actually see moving robots instead of simulated scenarios, you hear it crack when they crash, and you see the marks they leave on the playfield ...

4.2 Possible improvements

Before starting with this thesis, I merely knew that Lego Mindstorms existed and that it had something to do with Lego and robots. Though trying to accumulate Mindstorms knowledge fast, I missed some clues on the way that would have greatly improved things.

Mainly, I was too focused on what was there (Vision Command and accompanying software) rather than what I needed. If I was to do it all again, I would drop the VC software completely and process the camera image on my own. In fact, I've already done this with the filter driver, but there's still Vision Command's restriction to 8 regions. Besides, doing it all myself would also solve the broadcasting problem (see section 3.3.2.5) without the need to hack the Lego software. The code for talking to the IR tower could be copied from NQC (as the source is available, see [Baua]).

Using the original Lego firmware imposes some unnecessary restrictions too. The reason why I wanted to stick with it was to make it easy to copy and improve the environment. Now I know that it would have been worth it to switch to leJOS [Sol] or another firmware replacement. Not only do they allow to program in Java or "real" C, but they are also considerably smaller. This way, I could have easily avoided my constant struggle with out of memory errors.

Another big issue are the robots themselves. I focused largely on the software side of development, but I think modifying the robots in the right way could have worked wonders. However, I'm no specialist in Lego construction. When my son gets older, he will hopefully build better soccer robots for me ...

When it comes down to strategy, there's also a lot of work left. Most importantly, recognition of other robots should be improved. This could be done by having the robots send messages about their position, so it could at least be detected if the bumpers are pressed against a wall or another robot. Of course, this only works right when the world model of the sender is up to date. Also, "IR jam" could result when the robots all send at the same time. Next, calibration leaves much room for improvement too. Right now, there is a lot of running into walls to adjust the current orientation. I guess at least half of the walls could be skipped for the sake of fluid play and less collisions. Also, the floor pattern is hardly used yet. The bars are currently only taken as a direction indicator when finding an initial position, but they should also be employed to update the world model regarding the current position of the robot.

Finally, all I have implemented and that is reusable in other programs is contained in an NQC header file. If you want to use it, you have to write an NQC program yourself. However, it would be great to have a more intuitive user interface instead (or as a supplement). For example, a simple "strategy construction kit" is used in [LP00] to freely combine a set of predefined behaviours. Without such a tool, you can hardly motivate children or other people not experienced in programming to play around with it. An easy-to-use interface, on the other hand, would possibly allow using the soccer robots in educational or playing environments.

Anyway, I had a great time playing with Mindstorms and I can recommend it to everyone who's interested at least a little. Or as Lego likes to put it: just imagine ...

A NQC code for the soccer robots (excerpt)

```
// state constants
#define STATE_ALIGN                       1
#define STATE_ALIGN_TURNING_LEFT        201
#define STATE_ALIGN_TURNING_RIGHT       202
#define STATE_ALIGN_DRIVING               3
#define STATE_ALIGN_WAIT_FOR_TOUCH        4
#define STATE_ALIGN_FINISHED              5

// symbolic constants for ranges of light sensor values
#define MIN        0
#define MAX        1
#define INBETWEEN  2
#define UNDEFINED -1

// misc constants
#define NUM_SHADES 3

// direction constants
//  (must be exactly the numerical values used here)
#define NORTH    0
#define EAST     1
#define SOUTH    2
#define WEST     3

// light sensor variables
int shades[NUM_SHADES];
int currentSample;
int previousSample;
int currentLightness;
int previousLightness;
int oldSample;
int min;
int max;
int currentShade;
int grayBlackCount;
int blackGrayCount;

// state variables
int state;
int subState;
```

```
/**
 * Gathers information from the light sensor and remembers the
 * light shades found.
 * Doesn’t use any resources (except variables).
 */

void adjustShades() {
    int diffMark;
    int diff;
    int i;
    int minValue;
    int maxValue;
    int sample;

    // collect sensor information
    currentLightness = LIGHT_SENSOR;

    // search the nearest shade
    diffMark ||= currentLightness - shades[0];
    i = 0;
    if (diffMark > ADJUST_THRESHOLD) {
        diff ||= currentLightness - shades[1];
        if (diff < diffMark) {
            i = 1;
            diffMark = diff;
            if (diffMark > ADJUST_THRESHOLD) {
                diff ||= currentLightness - shades[2];
                if (diff < diffMark) {
                    i = 2;
                    diffMark = diff;
                }
            }
        } else {
            diff ||= currentLightness - shades[2];
            if (diff < diffMark) {
                i = 2;
                diffMark = diff;
            }
        }
    }

    // check if we have a new shade ...
    if (diffMark > SHADES_DIFFERENT_DIFF) {
        // ... yes we do
        //PlaySound(SOUND_UP);
        shades[currentShade] = currentLightness;
        currentShade = (currentShade+1)%NUM_SHADES;
                                        // ring buffer

        // find min and max shade
        // this is necessary since a min or max value might
        // have been thrown out
        minValue = 1000;
        maxValue = 0;
        for (i = 0; i < NUM_SHADES; ++i) {
```

```
            if (shades[i] < minValue) {
                minValue = shades[i];
                min = i;
            }
            if (shades[i] > maxValue) {
                maxValue = shades[i];
                max = i;
            }
        }

        // remember the sample
        //  (just in the form MIN, MAX, or INBETWEEN)
        if (currentLightness == minValue) {
            sample = MIN;
        } else if (currentLightness == maxValue) {
            sample = MAX;
        } else {
            sample = INBETWEEN;
        }

    } else {

        // adjust the shade
        if (diffMark > ADJUST_THRESHOLD) {
            // this is an expensive operation, so only perform it
            //  when significant changes occur
            shades[j] = (shades[i] + currentLightness)/2;
            //PlaySound(SOUND_CLICK);
        }

        // remember the sample
        //  (just in the form MIN, MAX, or INBETWEEN)
        if (j == min) {
            sample = MIN;
        } else if (j == max) {
            sample = MAX;
        } else {
            sample = INBETWEEN;
        }

    }

    // remember the previous lightness
    if (sample != currentSample) {
        oldSample = previousSample;
        previousSample = currentSample;
        currentSample = sample;

        if (currentSample == MIN) {
            if (previousSample == INBETWEEN &&
                oldSample == MAX) {
                ++blackGrayCount;
            } else if (previousSample == MAX &&
                       oldSample == INBETWEEN) {
                ++grayBlackCount;
            }
```

```
        }
    }
}


task adjustShadesTask() {
    while (true) {
        adjustShades();
    }
}


/**
 * Try to follow the border (right hand).
 * Uses Timer 1.
 *
 * State before:     doesn't matter
 * State after:      not changed
 * Substate before:  doesn't matter
 * Substate after:   many possibilities (don't rely on it)
 */

void followBorderRight() {
    if (subState == SUBSTATE_SWEEPING_RIGHT_FULL ||
        subState == SUBSTATE_DRIVING_RIGHT_FULL) {
        if (TOUCH_SENSOR_LEFT) {
            subState = SUBSTATE_SWEEPING_LEFT_FULL;
            setSweepingLeftFull();
            ClearTimer(1);
        } else if (TOUCH_SENSOR_RIGHT) {
            subState = SUBSTATE_DRIVING_LEFT_FULL;
            setDrivingLeftFull();
            ClearTimer(1);
        }
    } else if (subState == SUBSTATE_SWEEPING_LEFT_FULL ||
               subState == SUBSTATE_DRIVING_LEFT_FULL) {
        if (!(TOUCH_SENSOR_LEFT || TOUCH_SENSOR_RIGHT)) {
            subState = SUBSTATE_DRIVING_RIGHT_FULL;
            setDrivingRightFull();
            ClearTimer(1);
        } else {
            if (FastTimer(1) > 6) {
             // react after a certain time blocked
                if (subState == SUBSTATE_DRIVING_LEFT_FULL) {
                    subState = SUBSTATE_SWEEPING_LEFT_FULL;
                    setSweepingLeftFull();
                } else {
                    subState = SUBSTATE_TURNING_LEFT_FULL;
                    setTurningLeftFull();
                }
                ClearTimer(1);
            }
        }
    } else if (subState == SUBSTATE_TURNING_LEFT_FULL) {
        if (!(TOUCH_SENSOR_LEFT || TOUCH_SENSOR_RIGHT)) {
            setHalt();
```

```
            Wait(5);
            subState = SUBSTATE_SWEEPING_RIGHT_FULL;
            setSweepingRightFull();
            motorsFwd();
            motorsOn();
        }
    } else {
        subState = SUBSTATE_SWEEPING_RIGHT_FULL;
        setSweepingRightFull();
        motorsFwd();
        motorsOn();
    }
}


/**
 * Aligns orthogonally to a wall.
 * Uses timer 1.
 *
 * State before:     STATE_ALIGN or STATE_ALIGN_DRIVING (latter is
 *                   more optimistic when a touch sensor is first
 *                   triggered, but timer 1 must be cleared!)
 * State after:      STATE_ALIGN_xxx
 *                   (STATE_ALIGN_FINISHED when finished)
 * Substate before:  doesn't matter (but should be something
 *                   moving forward)
 * Substate after:   halt as soon as alignment is finished,
 *                   any other before
 */

void align() {
    if (TOUCH_SENSOR_LEFT && TOUCH_SENSOR_RIGHT) {
        setHalt();
        state = STATE_ALIGN_FINISHED;
    } else if (state == STATE_ALIGN) {
        if (TOUCH_SENSOR_LEFT && !TOUCH_SENSOR_RIGHT) {
            setTurningLeftFull();
            motorsOn();
            Wait(10);
            setHalt();
            ClearTimer(1);
            state = STATE_ALIGN_TURNING_LEFT;
        } else if (!TOUCH_SENSOR_LEFT && TOUCH_SENSOR_RIGHT) {
            setTurningRightFull();
            motorsOn();
            Wait(10);
            setHalt();
            ClearTimer(1);
            state = STATE_ALIGN_TURNING_RIGHT;
        }
        // if no sensor triggers, go on with whatever
    } else if (state == STATE_ALIGN_TURNING_LEFT) {
        if (FastTimer(1) >= 5) {
            setDrivingFwdFull();
            motorsFwd();
            motorsOn();
```

```
            state = STATE_ALIGN_WAIT_FOR_TOUCH;
        }
    } else if (state == STATE_ALIGN_TURNING_RIGHT) {
        if (FastTimer(1) >= 5) {
            setDrivingFwdFull();
            motorsFwd();
            motorsOn();
            state = STATE_ALIGN_WAIT_FOR_TOUCH;
        }
    } else if (state == STATE_ALIGN_WAIT_FOR_TOUCH) {
        if (TOUCH_SENSOR_LEFT || TOUCH_SENSOR_RIGHT) {
            ClearTimer(1);
            state = STATE_ALIGN_DRIVING;
        }
        // no timeout (yet)
    } else if (state == STATE_ALIGN_DRIVING) {
        if (FastTimer(1) >= 10) {
            state = STATE_ALIGN;
        }
    }
    // do nothing in case of an illegal state
    //  (saves program space)
}
```

B Bibliography

[Baua] Dave Baum. *NQC – Not Quite C*. http://www.enteract.com/~dbaum/nqc/.

[Baub] Dave Baum. *Using Mindstorms with a Macintosh*. http://www.enteract.com/~dbaum/lego/macmind/.

[Cap00] Ole Caprani. *RCX Manual*, 2000. http://www.daimi.au.dk/dArkOS/Vaerktoejer.dir/RCX.vejledning.dir/Vejledning.html.

[FvDFH96] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice, Second Edition in C*. Addison-Wesley Pub Co, 1996. ISBN 0-201-84840-6.

[Gas] Michael Gasperi. *Vision Command*. http://www.plazaearth.com/usr/gasperi/viscommand.htm.

[Juna] Andreas Junghans. *Lego Mindstorms*. http://www.fh-karlsruhe.de/~juan0014/mindstorms/index_en.html.

[Junb] Andreas Junghans. *OpenGL Grabber (Ogre)*. http://www.fh-karlsruhe.de/~juan0014/ogre/index_en.html.

[LEGa] The LEGO Group. *LEGO MINDSTORMS*. http://mindstorms.lego.com/.

[LEGb] The LEGO Group. *RoboLab – LEGO Mindstorms Sets for Schools*. http://www.lego.com/dacta/robolab/default.htm.

[LEG99] The LEGO Group. *LEGO Programmable Bricks Reference Guide*, Nov 1999. http://mindstorms.lego.com/sdk/index.html.

[LEG00] The LEGO Group. *RCX 2.0 BETA SDK*, Jul 2000. http://mindstorms.lego.com/sdk2/index.html.

[Log] Logitech. *Logitech QuickCam SDK*. http://developer.logitech.com/sdk/.

[LP00] Henrik Hautop Lund and Luigi Pagliarini. *Robocup Jr. with LEGO Mindstorms*, 2000.
ftp://ftp.daimi.au.dk/Staff/hhl/lund.RoboCupJr.ps.gz.

[Mic99] Microsoft Corporation. *Windows 98 Driver Development Kit (DDK)*, Jul 1999.
http://www.microsoft.com/ddk/ddk98.asp.

[Mic01] Microsoft Corporation. *DirectShow for Windows XP*, 2001.
http://msdn.microsoft.com/library/en-us/dshow/htm/directshow.asp?frame=true.

[Ove99] Mark Overmars. *Programming Lego Robots using NQC*. Utrecht University, Department of Computer Science, Mar 1999.
http://www.cs.uu.nl/people/markov/lego/tutorial.pdf.

[Pro99] Kekoa Proudfoot. *RCX Internals*, 1999.
http://graphics.stanford.edu/~kekoa/rcx/.

[RAWG01] Sandy Ressler, Brian Antonishek, Qiming Wang, and Afzal Godil. *Integrating Active Tangible Devices with a Synthetic Environment for Collaborative Engineering*. National Institute of Standards and Technology (NIST), U.S.A., Feb 2001.
http://www.itl.nist.gov/iaui/ovrt/people/sressler/tangible3.pdf.

[Rob] *RoboCup Official Site*.
http://www.robocup.org/.

[Sol] Jose Solorzano. *leJOS: Java based OS for Lego RCX*.
http://lejos.sourceforge.net/.