Bachelor thesis
Computer Science

Radboud University

Simulating NXT-robots

Author:
Giel Besouw
S448389

First supervisor/assessor:
Prof. dr. J.J.M. Hooman
`hooman@cs.ru.nl`

Second supervisor:
T.C. Nägele, MSc
`t.nagele@cs.ru.nl`

April 16, 2018

Abstract

High schools put ever more effort in riveting students for a higher education in science. The high school course NLT offers students a wide variety of subjects to spark interest in chemistry, physics, biology and computer science. This research intends to support the robotics module by investigating the possibilities to employ simulation software. We investigated the following research question: What are the possibilities to use virtual prototyping to support the NLT Robotics module? The answer to this question lies in software called Roboter Simulator. This allows the robot used in the module to be simulated. The program was tested for accuracy and modified where necessary according to a list of requirements which was set up in accordance with an interview conducted with one of the developers of the module.

Contents

Chapter 1

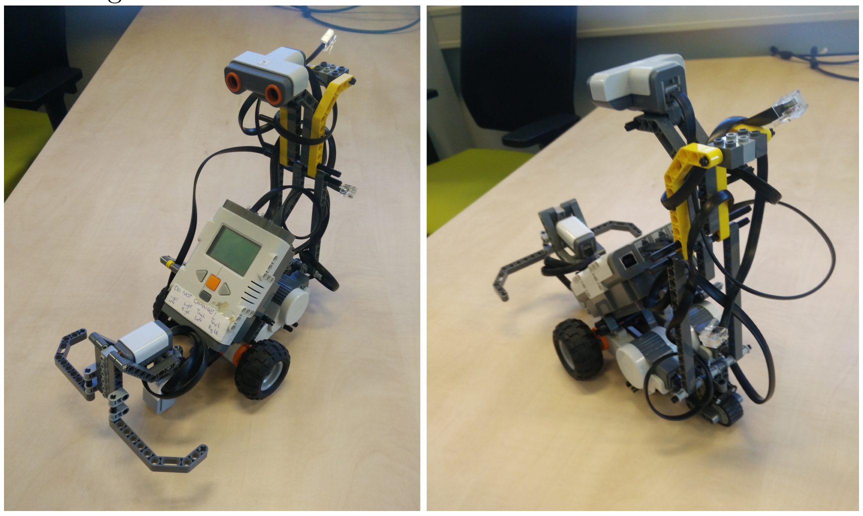
Introduction

This research is conducted in the context of the high-school course Natuur, Leven en Technology, translated literally to Nature, Life and Technology. Henceforth we abbreviate this to NLT. The course was introduced in 2007 to motivate students to choose for higher education in science. This course consists of many different modules which teach students about a variety of subjects in the field of biology, chemistry, math, physics and software science. To teach students about the field of robotics and ignite their interest in fields like electronics, programming and mechatronics, the NLT module Robotics was introduced [3]. In this module LEGO NXT kits are used. These kits come with a wide variety of parts to give people the freedom to build a large range of different robots. Parts range from wheels to gears to connection rods. To make the robot move three servomotors are included. These motors can operate at variable speeds and they are able to rotate with a precision of a single degree. It also features a number of different sensors, by default an NXT-kit contains the following sensors:

- Two touch sensors, to detect collisions or serve as physical buttons.
- One light sensor, to detect different colors and light-levels from a short distance. This can be used to detect whether the robot is driving over colored strips or to follow a path.
- One sound sensor, to discern between noise levels and tones.
- One ultrasonic sensor, to use echolocation to determine distance to objects in front of it.

In the NLT-module, students program pre-built robots which are not changed. Such a robot has two actuators, one touch sensor, one ultrasonic sensor and one light sensor installed. The actuators are connected to the front wheels, with a turning wheel mounted on the back of the robot. The touch sensor is serves as a bumper. The light sensor is attached behind the touch sensor facing the floor. In this configuration it can be used to detect color differences on the floor, which for example can allow the robot to follow a track. The ultrasonic sensor is mounted on the top of the robot. Figure 1.1 shows what the robot looks like.

Figure 1.1: Front and rear view of the robot used in the course



Using this robot the students have to do assignments by building a program and uploading this to the robot. Students are required to use BricxCC to program their robot [3, p. 11]. They have to use NXC, a programming language based on C made suitable for NXT-robots. The assignments start off simple, examples include: “Make the robot drive in the shape of an eight.” or ”Make the robot drive to the end of the arena and back”. This is supported by teaching students how to write understandable and expandable code and comments. Exercises get progressively harder; at the end of the course students are expected to solve problems like: ”Drive around the course reaching all checkpoints without bumping into any obstacles.” Especially the harder exercises teach students algorithmic thinking.

The NLT-module Robotics could be improved if robot hardware could be simulated. Most schools only have a few robots available for a group of twenty to thirty students. It is obvious that testing takes a lengthy amount of time. It should also be noted that it is impractical to assign homework since testing is impossible at home.

In general the simulation of hardware is referred to as virtual prototyping. Virtual prototyping is defined as simulating the behaviour of a certain object on a computer. In our study we limit ourselves to the simulation of NXT-robots. There has been little research on virtual prototyping with regards to educational robotics. Virtual prototyping is mostly used in product development to develop first versions which can be tested for early design flaws. These models can reduce development cost as they eliminate the need for a usually very expensive prototype [1].

Virtual prototyping is also frequently employed in the development of microcontrollers. In this context it is used as a tool to produce better quality systems for a couple of reasons. Firstly, by using simulated hardware beforehand, it can be adapted while software is in development to better support changing requirements as software changes. In the past, software had to be modified to compensate for poorly designed hardware, which had a detrimental effect on overall performance and adaptability. Secondly, the overall development time can be cut significantly by employing simulated hardware before developing a physical prototype. For instance, since the amount of physical robots is limited, testing could be sped up by letting students test their programs on a computer beforehand. Furthermore, trivial errors can be spotted and resolved much quicker [2].

When simulating the outside world, it is important to determine how much accuracy is needed. We define accuracy as how close simulation behaviour is to reality. Since the software is intended to run on school computers and students' laptops, a very intricate simulation environment is undesirable considering the limited power of the hardware it is running on. Furthermore, the goal of the simulation is to provide a general idea of how the robot will behave in the real world, not to verify details. Nevertheless, essential robot functionality should not be cut in favour of simulation speed.

Developing a new model requires time and effort. In all the aforementioned areas where virtual prototyping is relevant, model development requires expertise about the way the real-world equivalent behaves. Let's take as an example the microchip in a thermostat. Company C wants to develop a new microcontroller and decides to use a certain hardware platform. Furthermore C decides to employ virtual prototyping to speed up development. After a certain amount of time into development a revolution in electronics allows microchips to be constructed for a fraction of the price. These chips however, behave differently from the current ones and as such the model needs to be rebuilt. As expected someone with advanced knowledge on microcontrollers needs to construct a new model. While our field of study doesn't require very intricate details of the behaviour of actuators and sensors, changing the robot in ways like moving actuators and sensors, addition, removal or modification of transmissions or significant modification of weight or size of the robot invalidates the virtual prototype.

The goal of our research is to either find a viable solution for simulating NXT-robots in a school environment or we discover that development of new software is feasible given the means the Robotics course has got. Alternatively we consider this research a success if we can modify existing software to make it suitable for a high-school environment. Our current hypothesis points to slight modification of existing software to make simulation for educational purposes viable. To achieve this goal we devised a central research question with subquestions. These are as follows:

What are the possibilities to use virtual prototyping to support the NLT Robotics module.

a) What should the solution be capable of to support the Robotics module?

b) Are there existing solutions which can either be used or from which certain components could be used in a new program?

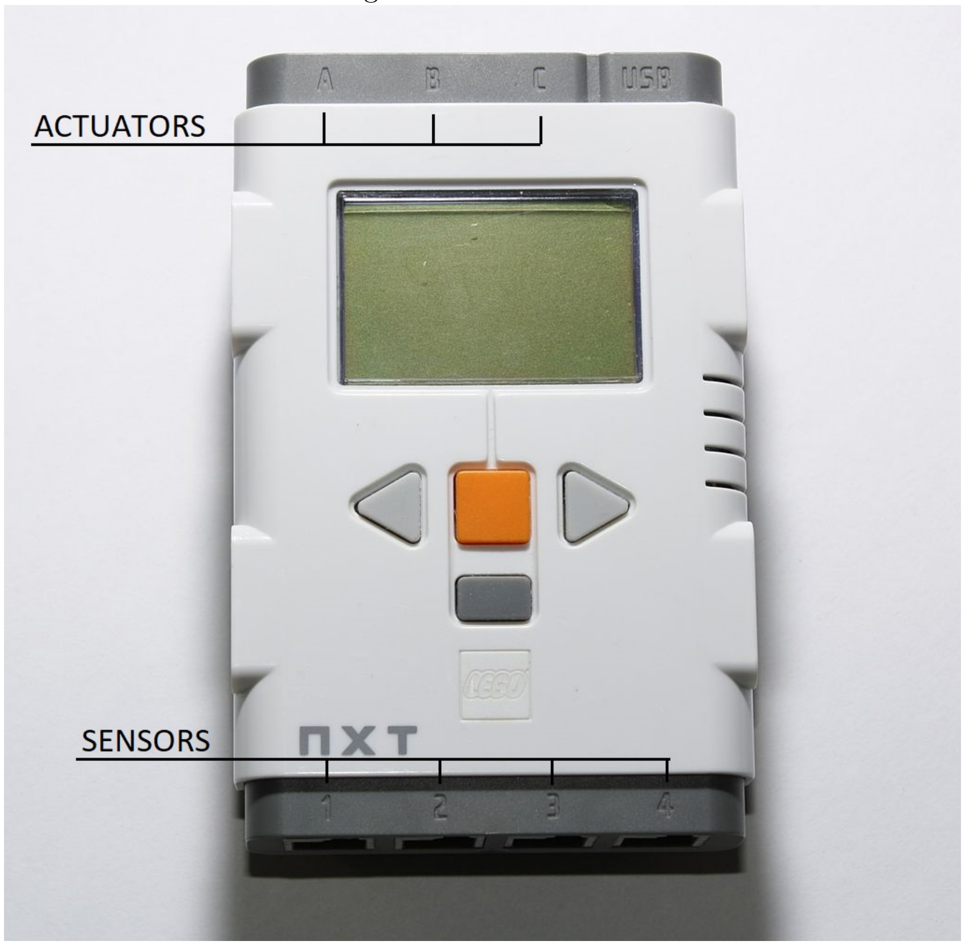
c) How accurate must the simulator be?

1.1 Preliminaries

Here the terminology will be explained in order to understand this paper

Actuators are the motors that make the robot move. The brick is the central computer for the robot. It handles sensor input, processes instructions and controls the movement of the actuators. The sensors and actuators are plugged into so-called ports. The brick has four in-ports, 1, 2, 3 and 4 which are used for sensors and three out-ports A, B and C which are used for actuators and lights. The brick is shown below in figure 1.2. NXT is the name of the robotic system. If we refer to the NXT-robot, we mean the physical robot students use. NXC is the programming language students write their programs in. This is compatible with the default operating system as used in the course. LeJOS is an alternative operating systems to be used by NXT-robots. Programs for LeJOS [10] are written in Java instead of NXC.

Figure 1.2: NXT Brick



Chapter 2

Research

2.1 Approach

We start with answering the first subquestion, that is, determining the requirements for this project. It's important to define these as quickly as possible since this will tell us what the most important aspects of the solution are.

We shall simultaneously investigate existing programs and see how they match these requirements. How we proceed depends on what programs we found. When we find candidate programs we need to test them for their viability in a school environment. Otherwise development of an early version of simulation software will be the next step. The goal, however, is not to develop a fully functioning virtual prototype, we mainly want to test for the viability of developing such a piece of software.

The requirements are formulated by conducting an interview with one of the developers of the module to figure out what they find important in simulation software. We ask teachers because then we can define requirements for a simulation environment that is both pleasant to work with and teaches students the most about the subject matter.

To answer subquestion c) we will run an experiment by comparing the behaviour of the real-world robot with the simulated one. Because we are dependent on the requirements to define some of our experiment variables, we can start with testing and writing code used in these experiments after the requirements are clear. The next course of action is defined by the previous steps. If we find a program which is sufficient we shall modify it where necessary. Otherwise a proof of concept will be written. We can include useful components we found in the other programs in this prototype. The final steps are finishing the thesis, writing an action plan for future research following up on this thesis and presenting this thesis.

Table 2.1: Planning

#	**Activity**	**Deadline**	**Expected Result**
1	Find existing software for simulating NXT-robots	12-11-2017	We will be able to find a number of existing solutions.
2	Define requirements	26-11-2017	We will be able to make a list of requirements.
3	Define tests and write programs to determine accuracy	26-11-2017	Programs and test parameters will be written. Chapter on experiment will be written.
4a	Determine accuracy of found solution	8-12-2017	We will find the found solution to be accurate
4b	Determine which parts of the found solution can be reused	8-12-2017	-
5a	Determine where the program needs to be modified	17-12-2017	Program is ready for use as-is.
5b	Write prototype	1-1-2018	Modified program can be tested in schools.
6	Write action plan for finishing the project.	1-1-2018	Action plan is ready
7	Finish writing thesis	8-1-2018	Final version is ready
8	Give presentation	17-1-2018	Presentation has been done

2.2 Planning/Timetable

The project planning can be found in table 2.1.

2.3 Requirements

We have conducted an interview with one of the teachers responsible for the development of the module Robotics to determine what needs to be implemented to provide the most benefit to the module. We have listed the requirements for the simulation environment below ordered from most to least important:

1. Make the simulator and the real robot accept the same code.
2. The software must be compatible with Windows PCs.
3. Make the simulator as accurate as possible. This means a difference in execution time of at most 10 percent. This is further explained in chapter 4.

4. Allow lights and sounds to be simulated
5. Allow text messages to be printed
6. Add support for a colour sensor
7. Add support for the next generation of LEGO robots, the EV3 system.

The requirements were determined in such a way that the resulting program would help students the most with doing the exercises. To make our simulator as accurate as possible, we shall take two steps. Firstly we will conduct the aforementioned experiment to determine where differences in the behaviour of the real robot and the simulation lie. The second step is simply changing the program to remove most, if not all, of these differences. The way sounds and lights will be implemented depends on the amount of time we have left after completing the previous tasks. If we have plenty of time we will add functionality to actually show lights and make sounds. Should this not be the case we will reduce this to messages saying that a certain light or sound has been activated. Implementation of text messages will be developed concurrently with sounds and lights since they work the same way. Support for colour sensors will probably not be developed since it will require a large investment of time. Adding support for the EV3, which is the next generation of LEGO robots, is future work.

Chapter 3

Existing Tools

We scoured the internet to find software which can simulate NXT-robot behaviour. We came upon three open source solutions and two closed-source commercial solutions.
Open source:

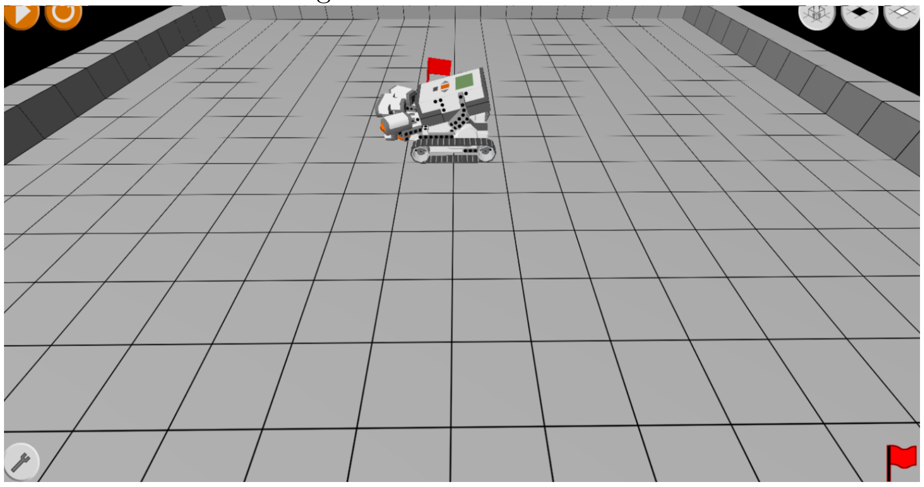
- Roboter Simulator [5] [11]
- RobotSim [6]
- NXCSimulator [7]

Closed source:

- Robot Virtual Worlds [4]
- Virtual Robotics Toolkit [9]

3.1 Roboter Simulator

Figure 3.1: Roboter Simulator



The first piece of simulation software we came upon is Roboter Simulator [5]. It features 3D graphics with a simple arena made up of blocks. It allows the users to raise, lower and color these blocks to create an obstacle course. The light and touch sensors interact with these modified blocks to create dynamic behaviour. The shape and actuators of the robot are always the same and sadly Roboter Simulator offers no way for the user to modify these settings. The developers of this program built the robot like a standard "car-bot". Out-port B and C go the left and right wheels respectively. By default in-ports 1 and 4 are connected to the left and right touch sensors, in-port 2 is connected to the light sensor and lastly in-port 3 is connected to the ultrasonic sensor which is attached on the front of the robot. The simulator allows the user to modify these sensor settings by changing their type, their rotation in perspective to the brick and whether the sensor faces forward or downward. Roboter Simulator is integrated with BricxCC to allow easy uploading of programs to the simulator.

First we shall list pros and cons of this program to get a general idea of the suitability for the NLT-module.

+ 3D-environment allows for good visualization.

+ The fact that Roboter Simulator is open source will allow for easy modification of the program or repurposing of components should the complete package prove to be not up to our requirements.

+ Integration with BricxCC allows for easy testing.

+ The layout of sensors is different from the layout of the robot used in the NLT-module. However this layout can be modified to match our robot.

– The block-based environment doesn't allow for precise obstacle courses.

– Sensors do not seem to work.

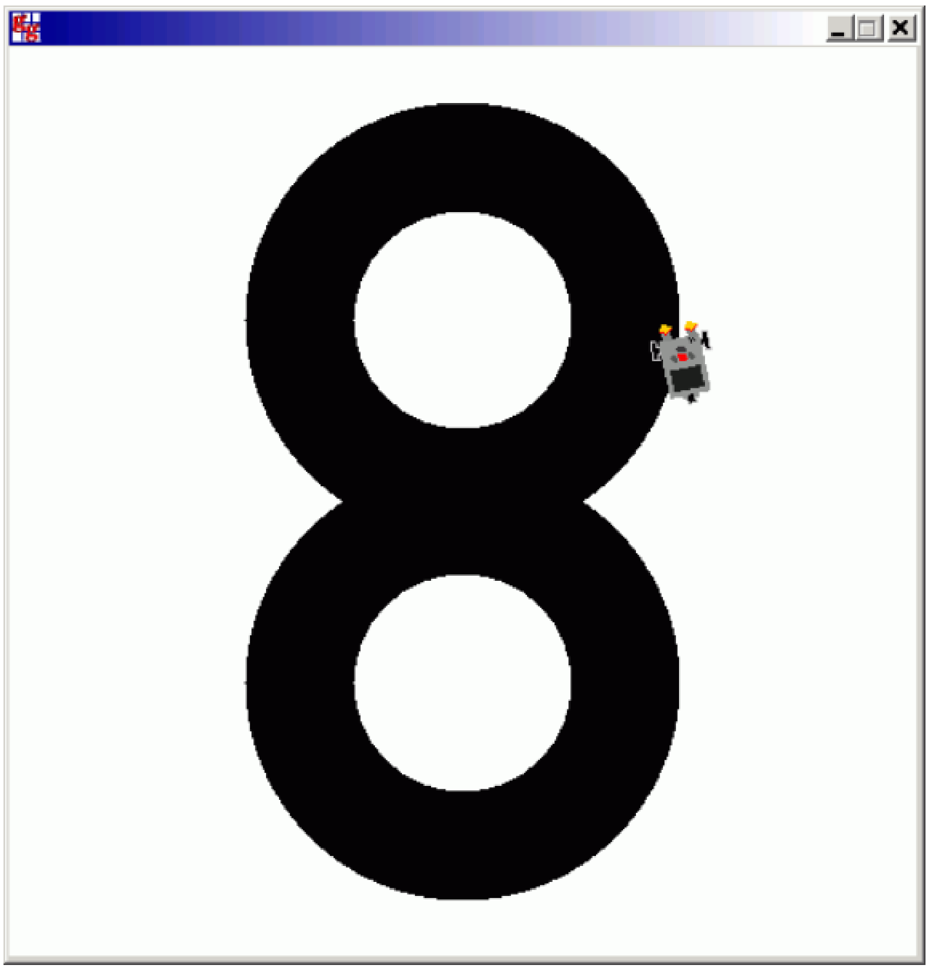
– There is no colour sensor support.

While this might seem like a convincing list of pros and cons, the lack of sensor functionality is significant disadvantage since these are essential for nearly all robot programs. The situation now is that the robot simply plows into the wall and keeps going even if the programming linked to a sensor-signal would dictate otherwise. This means that Roboter Simulator is unsuitable for simulating for NLT-students unless settings can be modified to make sensor functionality work. Alas only simulating programs which do not make use of sensors is not very exciting and not very useful considering the assignments students have to do.

A closer inspection of the source code reveals that the issue Roboter Simulator suffers from stems from the interpreter. Sensors do work, however the interpreter present in the simulator ignores sensor-handling the way it is taught in the NLT-module.

3.2 RobotSim

Figure 3.2: RobotSim



The second piece of simulation software we found is RobotSim by Aegidius Plüss[6]. It is an open source Java and Python library. It allows for users to write Java programs for the NXT-brick. The simulation needs to be setup by writing commands in a simulation class. In this same class, settings regarding the playing field can also be added. The paragraph below contains a more detailed explanation on how to setup a simulation. This simulation class can be executed to render a 2D environment with a robot and a field. The robot will perform the commands; the actions associated with these commands are visualized in real-time.
The programming of simulation classes for RobotSim is somewhat different from how it is done in BricxCC. The simulation classes are written in Java along with the code the robot needs to run. Another difference is that one

does not tell an individual actuator what to do when using RobotSim. To move the robot one tells it to go forward, left, etc. This combination of actuators is called a gear. This is different from BricxCC programming in the sense that BricxCC requires users to control individual actuators to perform actions. If one would want the robot to turn around its center, you would program an actuator to go forward while the other turns in reverse. Sensors are programmed in the same way as in BricxCC except for the syntax. At some point in the program the sensor is activated, after which its state can be extracted. In the case of a touch sensor this can be either 0 or 1, reflecting whether it is pressed or not. When programming a RobotSim simulation the state of a touch sensor can be checked with the isPressed function.

First we shall list pros and cons of RobotSim to get a general idea of the suitability for the NLT-module.

+ The arena can be modified easily and extensively; obstacles of many shapes and sizes can be added and it features the ability to color the floor.

+ Because of the way the software is built, it is possible to simulate multiple robots at once.

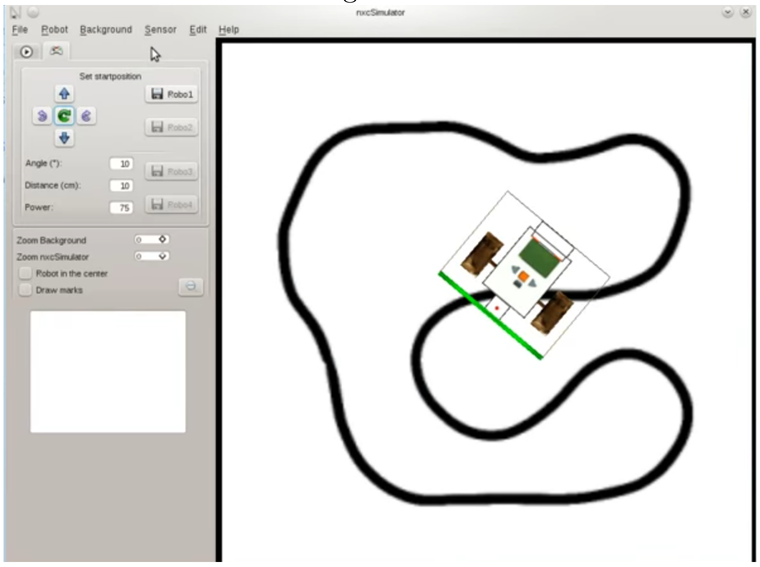
– This software only simulates programs written for leJOS which means that the NLT-module needs to be changed considerably to make this software usable.

– Regardless of the language the original program was written in, to simulate a program it has to be translated to make use of gears in order for it work with RobotSim.

RobotSim will not be experimented upon for accuracy since there are two problems which prevent NXTSim from being a viable solution. Firstly the code is different from the code that a robot running leJOS accepts. As was mentioned earlier NXTSim makes use of gears. leJOS does not support this way of implementing actuators. Even if the course would be changed to have the robots make use of leJOS, students' code would still need to be modified to make use of NXTSim. The second problem is the need for arenas to be built programmatically. The developers of the course would need to create every arena beforehand, ask of students to program arenas or teach every NLT teacher how to implement arenas. Every approach has its disadvantages. The first alternative is very inflexible, the second is highly likely to fail as students are only just introduced to programming and the third is impractical and time-consuming.

3.3 NXCsimulator

Figure 3.3: NXCsimulator



Angle (°): 10
Distance (cm): 10
Power: 75
Zoom Background 0
Zoom nxcSimulator 0
Robot in the center
Draw marks

NXCsimulator for NXCeditor[7] is the third option to simulate NXT-robots. Just like the first option we discussed, this program is an extension of an existing editor. NXCeditor supports programming NXT and EV3 bricks in C, C++ and NXC, the same language used when working with BricxCC. The simulated robot can be highly customized in the settings menu; the position and size of the wheels can be modified and the position, type and orientation of the sensors can be changed. First we shall list pros and cons of this program to get a general idea of the suitability for the NLT-module.

+ The robot is highly customizable

+ Integration with NXCeditor allows for easy testing.

+ The usage of bitmaps to represent obstacle courses allows for easy customization of the arena.

– Since NXCeditor is currently not being used by the NXT-module, some changes have to made. However, the interface and functionality is largely the same so the transition shouldn't be too difficult.

– This software only runs on Linux or a Linux-based virtual machine.

– NXCsimulator is highly unstable

Getting NXCsimulator to run is a significant challenge since it is build on the Gambas3 [8] platform. This is an object-oriented dialect for BASIC.

The difficulties reside in a couple of factors. Firstly Gambas-code cannot be compiled to run on Windows machines. This presents problems with rolling out software in schools as most machines run Windows. Secondly the software is highly unstable, segmentation faults are common.

3.4 Commercial Solutions

We found two commercial NXT simulation software packages. After a short look we found out that both of these solutions are closed source and rather expensive. Still, we shall add them to our list for completeness sake. Robot Virtual Worlds is the first commercial software package we found. It features a complete 3D environment and the built-in exercises helps students learn programming. It is available for multiple robotic platforms [4].
The second commercial package is called Virtual Robotics Toolkit. Virtual Robotics Toolkit comes with a 3D environment just like Robot Virtual Worlds. The most important feature however is the physics engine in conjunction with the ability to import robots from LEGO Digital Designer. This means that any type of robot, not just the Tribot, can be simulated with this program. This allows for a lot of flexibility with assignments since the students are not restricted to one type of robot. Sadly this package only supports standard block-based programs; this is the same type of programming as is used by default for NXT-robots.

3.5 Conclusion

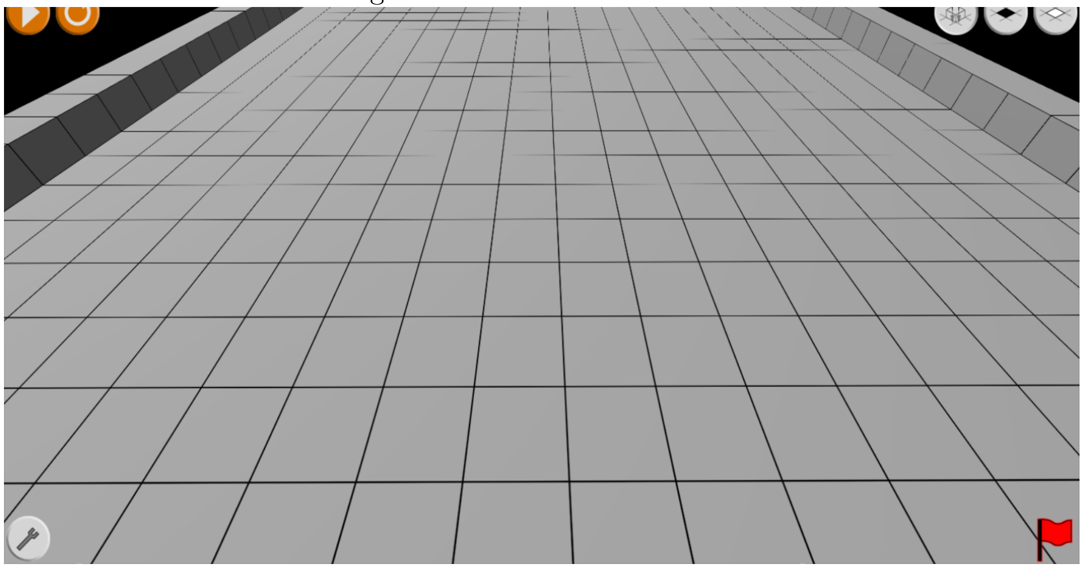
It turns out Roboter Simulator is the only viable solution and is the only one we will use for accuracy experiments. NXCsimulator simply does not work on Windows. RobotSim isn't a fleshed-out program. This is compounded by the fact that it does not support NXC-code; the primary benefit of simulating robots, namely saving time, is nullified by all the extra steps involved in running a simulation with RobotSim.

Chapter 4

Accuracy Experiments

One of our requirements for this project is accuracy. Our way of getting an idea of accuracy is with experiments. To determine whether the simulation software we found online is suitably accurate with regards to the real-life behaviour of our robots, we shall run a number of tests. These tests are aimed at the movements of the robot as these are essential to a robot. The actuators define the movement of the robot. To find out how the movement of the robot and the simulator differs we need to find out the difference between actuators of robot and simulation. This difference can best be measured by looking at the difference in speed. The experiments are set up so that the robot and simulation both drive the same distance. By recording execution time we can find out how fast both are. Since we cannot measure distance with a ruler in the simulator, we have to find which unit is equivalent to a centimeter in real life. The only value in the simulator which is comparable to real life is the size of the robot. The simulated robot is approximately 2 by 2 blocks in size. The entire arena of the simulation is 20 by 20 blocks in size. This equals 10 by 10 robots. The real life version has a size of 27.5cm. This gives us an arena size of 275cm. Minus the blocks on all sides a functional arena of two and a half meters remains. In figure 4.1 the arena is shown.

Figure 4.1: The arena



We have made two programs and determined a norm for the differences between the robot and the simulation; the average may differ at most 10%. The contents of these programs can be found in appendices A and B. Each program has two variants; one is used by the robot and one is used by Roboter Simulator. Appendix A is a program which instructs the robot to drive forward until it hits a wall. Appendix B instructs the robot to spin around its axis for a set amount of spins. We have made two variants for the program. Variant A is used by the robot and makes the robot show the execution time after it has finished running. Variant B is used by the simulator. The program of appendix C is the same as the one in appendix B only in the other direction. It makes the robot turn before running the real test. This is done to position the simulated robot 100cm from the wall which is the same distance as the real robot drives. These programs serve to answer the following questions:

- Is the speed of the robot the same as that of the simulated robot?
- Are the turns made by the real robot the same as the ones made by the simulated robot?

The following environmental conditions will be used to determine the consistency of behaviour of the real-life robots:

- Different power levels for motor input. See section 4.1.
- Different battery levels, we shall test the robot at full, three-quarter full, half, quarter full and nearly empty battery life. See section 4.2.

- The surface on which the robot is driving. We shall test on a paper obstacle field, a wood platform and linoleum flooring. See section 4.3.

Our experiments will be performed in multiple iterations. Every iteration we shall modify the power constant of variant B of our program to correct inaccuracy found in the previous iteration and run it again. This will be repeated until the difference between variant A and variant B falls within the margin of 10%. The goal is to find what value relevant constants have to be changed to make to Roboter Simulator accurate. This will be explained more thoroughly in chapter 5. All experiments follow the same pattern. The program will be run 10 times and the times in the simulator will be measured with a stopwatch. The times of the robot will be measured with an internal clock. The median of the 10 executions is taken so any outliers don't influence the result. Times are shown in tables; The unit of measurement for the other results are second:hundredth of a second. Firstly the experiments on environmental conditions will be shown.

4.1 Power level

We mentioned earlier that we are going to test the influence of different factors on the behaviour of the robot. First the difference in execution time will be measured with different power levels. Ideally we will find that execution time decreases linearly with increase in power. The first run will determine the execution time for both simulator and robot at a power level of 50%. This is the same as the power level of other experiments. The program in appendix A is used. The results are shown in table 4.1. The unit of measurement for this and the other results are second.hundredth of a second.

Table 4.1: Power experiment iteration 1

	Robot	Simulator
	06.24	01.66
	06.20	01.70
	06.25	01.62
	06.26	01.56
	06.27	01.55
	06.26	01.53
	06.25	01.68
	06.27	01.61
	06.30	01.55
Median	06.26	01.62

We have reduced the power to 25%. After running the experiment again we found the results shown in table 4.2.

Table 4.2: Power experiment iteration 2

	Robot	Simulator
	12.82	03.16
	12.94	03.13
	12.62	03.42
	13.01	03.33
	12.97	03.28
	13.09	03.37
	13.27	03.39
	13.03	03.11
	13.19	03.23
Median	12.97	03.23

From our results we can see that halving the power about doubles the simulation time. The real robot takes just over twice as long to complete the same program, while the simulator uses slightly less than twice the amount of time. We will now increase the power to 100%. We expect the average time to be half as long as the time shown table 4.1. The results of this iteration are shown in table 4.3.

Table 4.3: Power experiment iteration 3

	Robot	Simulator
	03.03	00.72
	03.05	00.89
	03.04	00.84
	03.01	00.86
	03.03	00.87
	03.04	00.80
	03.03	00.98
	03.06	01.05
	03.04	00.99
Median	03.03	00.87

We can see from the results that the speed of the robot scales linearly with power.

4.2 Battery level

In contrast to the simulation the real robot runs on a battery. We were unable to find literature comparable to this one where the influence of battery level on actuator performance was tested. Since the simulator's precision is one of the most important requirements we want to figure out the whether the battery level affects performance.

The robots will once again execute the programs described in appendix A, this time with reduced battery. battery levels are reduced to 75%, 50%, 25% and 5%. At each level we executed the programs and kept track of the time. The results are shown in table 4.4.

Table 4.4: Battery Experiment

	75%	50%	25%	5%
	06.18	06.27	06.32	06.43
	06.22	06.29	06.28	06.39
	06.32	06.28	06.28	06.40
	06.24	06.31	06.32	06.40
	06.23	06.32	06.32	06.43
	06.25	06.29	06.33	06.46
	06.27	06.24	06.35	06.46
	06.26	06.30	06.34	06.50
	06.26	06.31	06.36	06.49
Median	06.25	06.29	06.32	06.46

The results show us that the difference between battery levels is negligible. The execution time difference between 100% en 5% is two-tenth of a second compared to a total execution time of over 6 seconds. This is well within our margin of 10% and therefore we do not deem it necessary to take the battery level into account for simulation.

4.3 Surface

As mentioned earlier we wish to measure in what way the surface affects the performance of the robot. Since no research has been done in this area with regards to NXT robots a short experiment has been done to find out whether this influence is significant. We shall execute the program described in appendix A on the most common surfaces. These are paper, wood and linoleum flooring. One might wonder in what situation these surfaces come into play. The arena which the light sensor interacts is printed on paper. The platforms on which the robots drive are made out of wood; when the paper arena isn't necessary the surface is thus wood. The floor is used when testing intermediate programs. The surface for every test other than the ones described in this section are on wood. Test results are shown in table 4.5.

Table 4.5: Surface experiment results

	Paper	Platform	Floor
	06.07	06.03	06.39
	06.10	06.03	06.38
	06.15	06.06	06.41
	06.11	06.04	06.43
	06.15	06.06	06.45
	06.21	06.06	06.41
	06.16	06.09	06.45
	06.17	06.06	06.40
	06.15	06.08	06.43
Median	06.15	06.06	06.41

As one can see the difference between the paper playing and the wooden platform is negligible. It easily falls within the 10% accuracy margin. The difference with the floor is more significant but still falls within the margin. Hence no modifications are necessary.

4.4 Speed experiment

The results from the experiment show that the robot behaves consistently under different conditions. We will use the program of appendix A to determine the difference in speed between robot and simulation. The robot uses variant A while the simulator uses variant B. The robot's execution time will be measured with a built-in stopwatch. The results of our tests are shown in the table 4.6.

Table 4.6: Experiment 1 iteration 1

	Robot	Simulator
	06.18	01.90
	06.03	01.88
	06.23	02.00
	06.09	01.81
	06.41	01.83
	06.06	01.72
	06.13	01.91
	06.41	01.83
	06.82	01.78
Median	06.23	01.83

Since we are seeing considerable differences between our simulation and robot, we decided it best to reduce the speed of the simulation. On average the simulation time is a about a third of the real execution time. We shall therefore cut the power by two-thirds. We redid the experiment with the a modified constant pwr. Constant pwr in variant B has a value of 16. The results of the second iteration are shown in table 4.7.

Table 4.7: Experiment 1 iteration 2

	Robot	Simulator
	06.18	05.18
	06.03	04.70
	06.23	05.01
	06.09	05.03
	06.41	05.05
	06.06	04.92
	06.13	04.93
	06.41	04.90
	06.82	05.07
Median	06.23	05.01

The simulated robot drives about as fast as the real robot. But the norm of 10% is not achieved. Hence pwr will be changed to 12. The next iteration of the experiment yielded the results shown in table 4.8:

Table 4.8: Experiment 1 iteration 3

	Simulator	Robot
	06.32	06.18
	06.37	06.03
	06.35	06.23
	06.47	06.09
	06.31	06.41
	06.35	06.06
	06.53	06.13
	06.40	06.41
	06.21	06.82
Median	6.35	06.18

At this point we consider the differences between the real world and simulation negligible since our norm is satisfied. We now know that we have to reduce the speed of Roboter Simulator to 25% of its original value. In the next chapter we will show what constant exactly was changed.

4.5 Turning experiment

We continue with the next part of the experiment. We have to find out what the difference of turn speed is between the simulator and the robot. We use the program defined in appendix B and C to test the turn speed. The program of appendix C is the same as the one of appendix B only in the other direction. Just as with the first iteration we shall measure and compare execution time to determine accuracy. This program makes the robot spin around its axis. We have measured the time it takes for the robot to spin five times. Just as with the speed experiment the robot's time is measured by the robot itself while the simulator is timed with a stopwatch. Both left turns and right turns were tested. The times are shown in the table below:

Table 4.9: Experiment 2 iteration 1

	Robot left	Robot right	Simulator left	Simulator right
	10.97	11.15	21.70	21.62
	10.89	10.71	21.63	21.63
	10.92	11.05	21.53	21.53
	10.87	10.79	21.57	21.68
	10.98	11.07	21.54	21.49
	10.99	11.06	21.43	21.57
	10.97	11.13	21.67	21.67
	10.94	10.89	21.44	21.63
	11.01	10.87	21.72	21.67
Median	10.97	11.05	21.63	21.63

As can be seen the real robot spins twice as fast. The turn speed constant in Roboter Simulator was increased by 100% to compensate for this difference. The exact change is described in the next chapter.

Chapter 5

Modifications

In this chapter we shall explain which components Roboter Simulator contains and the changes we shall make to these components to fulfill the requirements. Only parts of the actual Roboter Simulator software will be considered, i.e. external libraries used by Roboter Simulator will be left out. We shall first provide a description of the classes that make up the program. These descriptions feature a short explanation of the function of the class and whether or not it will be modified. The second part describes in what way we modified classes; how they used to behave and how they behave now. We have put these classes in categories; classes categorized as networking or rendering classes will not be modified. The rendering classes work as they should and networking functionality is not part of our requirements. We have listed these classes for completeness.

5.1 Behaviour

Controller.cpp. This class brings everything together. The camera and controls are handled by this class. It also defines things like the size of the arena. The functions defined here make the application behave the way it should. It holds a variable which defines the turn speed of the robot. If we find the turn speed to be differing too much from the real-world behaviour of the robot, we shall modify it in this class.
Environment.cpp This class handles the arena. Although the size of the arena is defined in Controller the actual functionality is defined in this class. This class works fine as is, hence we will not change it.
Interpreter.cpp and helper classes. These classes take code as input to transform these into instructions for the robot. We need to modify these classes to make the robot and simulator accept the same code.
Motor.cpp. The name implies its function, this class takes care of the functionality of the motors of the robot. As was mentioned in the chapter on the experiment, we modified the speed constant to make the real robot

and the simulated one equally as fast.
Robot.cpp. This class defines the robot and all of its properties. Properties such as wheel position, sensor position and general shape. We have modified this class to make the simulated robot resemble the real robot more.
RobotSpeaker.cpp RobotSpeaker implements sounds which the real robot can also produce. We do not have to modify this class as there are no specific requirements with regards to sounds other than basic implementation.
Simulation.cpp. Simulation handles the robot interacting with the environment. Things like the robot running into a wall is handled by this class. We did not modify this class.
SoundBuffer.cpp. This class makes sure the sounds the robot makes are not played concurrently. As we mentioned in the paragraph on RobotSpeaker, there are no specific requirements with regards to sound other than the implementation of it.
SoundController.cpp. This is the controller-class for everything sound-related. Just as Controller.cpp for the whole program, this class brings all the sound-classes together and makes it into a functioning whole. Again, this component will not be modified.
System.cpp. System.cpp is a little less obvious; it simulates the NXT-brick. On first sight this seems unnecessary because the Brick handles sensor input and motor output. However the brick is also capable of basic math, control and data instructions. In short, it is capable of executing basic computer programs. Since this class emulates the brick which works as it should, this class will not be modified.

5.2 User Interface

EnvironmentEditor.cpp. This class gives the user the ability to edit the arena on-the-fly by raising, lowering and coloring blocks. This class might be needed to be modified to allow users to color the blocks more than just white, grey and black. Should we have time to implement color sensors, colored blocks will need to be available as well.
SensorConfigurationScreen.cpp. This class builds part of the UI, namely the buttons and dials with which to modify type, orientation and direction of the sensors. Should we have enough time to implement color sensors we will have to modify this class as well.
UI elements. UIButton.cpp, UIIcon.cpp, UIKnob.cpp and UIRadioGroup are the elements that make up the user interface of the program. These classes only handle the interaction of the user with these elements, what happens afterwards is passed to other classes. We will need to add text messages according to the requirements, so another class needs to be added to this set.
UserInterface.cpp This class groups the elements mentioned above into

a single class. This class executes what should happen when a certain UI-element is interacted with. This class needs to be modified to allow for text messages to be printed on-screen.

5.3 Networking

We will not modify this category of classes.
Client.cpp. This class handles client functionality for networking with this program.
NetworkInterface.cpp. This is an interface for the networking part of the software.
NetworkPacket.cpp. This class implements packets for the networking part of the software.
Server.cpp. This class handles server functionality for networking with this program.
ServerBrowser.cpp. ServerBrowser implements the window which allows clients to search for servers running this software.

5.4 Rendering

We will not modify this category of classes.
Drawer.cpp. The Drawer class handles rendering of 3D graphics on screen. It holds all the necessary code to draw the robot and arena. The program uses the SDL-library to support 3D graphics, which is used in this class.
EnvironmentDrawer.cpp. This is a support class for Drawer, it handles drawing the arena.
Model.cpp. Helper class for Drawer. It is used to translate vertex data suitable for SDL.
RobotDrawer.cpp. This is a support class for Drawer, it handles the drawing of the robot.
Texture.cpp. Texture handles the texturing of the rest of the 3D environment i.e. the robot and the arena. It's a helper class for **Drawer.cpp**.

5.5 Changed constants

In the previous chapter we performed an experiment to determine the differences between the movement of the simulator and the robot. The first change we made to Roboter Simulator is modifying the variable powerToSpeedRatio from 2.5 to 10.0 in **Motor.cpp**. This had the effect of making the speed of the robot close to the speed of the simulated robot. As a result from this change the turn speed of the simulation needed to be doubled. We fixed this by modifying the turnSpeed variable in Controller.cpp from 3.0 to 6.0.

5.6 Future changes

This section serves as a guideline for future changes. In section 2.3 we listed a number of requirements for the simulation software we found. We shall describe what requirements still need to be implemented and how we think this can be done.

The first part we would add are lights and sounds. We prioritize this requirement because it is the most important unimplemented requirement behind requirement 1. The reason we choose lights and sounds over requirement 1 is the ease of implementation. Furthermore the overall benefits of implementing requirement 4 outweigh the benefits of implementing requirement 1. The only reason requirement 1 is not fulfilled is because of the difference between the way sensors function. Everything else is programmed the same. We would implement requirement 5 afterwards since this is the next in the list of requirements. The next requirement to implement is requirement 1. requirement 6 requires a lot of work to implement. To have a colour sensor the arena must first be able to be coloured something else than just black, grey and white. After implementing requirement 1, colour sensors should be implemented.

The authors of the original program[11] stated their opinions on how they would improve Roboter Simulator. Regarding the scope of Roboter Simulator at the time of implementation, the authors did not implement the complete bytecode. This simply means that not all NXC operations are supported by the simulator. This has became clear in Roboter Simulator when certain ways of interacting with sensors did not function. The authors mentioned that in future work it an avenue of improvement would be expanding the bytecode. We agree as it removes the need for students to change their code to get it to run, albeit it small changes. The different way of implementing sensors for Roboter Simulator comes to mind. A further improvement proposed by the authors is the usage of a more advanced physics engine. We believe this is not necessary because it serves to improve the accuracy of the simulator. Accuracy can be assured however, with modifications to constants specific to the robot, especially considering the assignments students have to do. Another improvement the authors propose is adding more features to the arena. They give the example of ramps to allow the robot to go to a higher point in the arena. For the same reason as with the improved physics engine, the NLT course does not require these kind of features.

Chapter 6

Conclusions

We set out to answer the question whether we could support the NLT module Robotica. This question was supported by a number of sub-questions. These questions are as follows:

a) What should the solution be capable of to support the Robotics module?

b) are there existing solutions which can either be used or from which certain components could be used in a new program?

c) How accurate must the resulting simulator be?

The first sub-question was answered by conducting an interview with one of the developers of the NLT module Robotica. The most important requests were to assure the simulator is accurate, students' code doesn't have to be modified for the simulator to accept it and the software runs on Windows. The answers to this question gave us the criteria which helped us answer the second sub-question. We found that Roboter Simulator is good enough to be used by students in the module Robotica, albeit with modifications. We answered the third subquestion by combining an experiment with the aforementioned interview. We compared the run-time of the simulator with the robot's run-time. In the interview we agreed upon a margin of 10 percent. At first the run-times differed by a factor of three. We modified Roboter Simulator to reduce the difference and reran the experiment until the difference fell within the margin.

In the end we succeeded in fulfilling the most important requirements to a large degree, in this sense we can say we are able to support the NLT-module with simulation software. We have summed up all requirements below with a short review on the extent to which each requirement is fulfilled.

1. Make the simulator and the real robot accept the same code. We mentioned earlier that the way students program sensors is different for the simulator and the real robot. In short there are two ways to program the sensor but one way does not work in conjunction with Roboter Simulator. This is the only difference between the code that is accepted by the NXT robot and the simulation. Still, strictly speaking the requirement is not completely fulfilled.

2. The software must be compatible with Windows PCs. This requirements is fulfilled as Roboter Simulator runs natively on Windows machines.

3. Make the simulator as accurate as possible. This means a difference in run-time of at most 10 percent. We ran a series of experiments to determine how accurate Roboter Simulator was. Roboter Simulator required changes to a number of constants in order to have it fulfill the norm. These were succesfully implemented; the requirement is fulfilled.

4. Allow lights and sounds to be simulated. As was mentioned in the section on the action plan, it proved to be very difficult to get the source-code to compile. This resulted in not having enough time to implement additional features. As such this requirement is not fulfilled.

5. Allow text messages to be printed. See requirement 4.

6. Add support for a colour-sensor. See requirement 4.

7. Add support for the next generation of LEGO robots, the EV3 system. This requirement is deemed the least important. The developer of the Robotica course mentioned EV3 support doesn't become relevant in the near future. Because of this the requirement is not fulfilled.

Chapter 7

Discussion

Here we will discuss what we learned from this project, whether we deem the project to be a success and how we would improve this kind of research in the future.

Modifying open source software was not as easy as we initially thought. Getting the source code to compile was a challenge but with the help from the initial developers of Roboter Simulator we managed to implement some of the proposed changes. We implemented fewer features than we expected to. As mentioned earlier, modification of existing software was far more time-consuming than we initially thought.

The results from our experiment were not as we initially expected. The original version of the simulator moved three times as fast as the real robot. We do not know why the original creators tuned the simulation to be this much faster. It was equally as surprising after modifying simulation speed to better match reality, that the turn rate of the simulated robot was slower. A possible explanation might be that the robot on which the simulation is based, had it's wheels further apart than the robot used in the course.

There are certain limitations to our research. Although we consider it a moderate success with regard to our requirements, we have done no research on the actual usability by high-school students.

We suggest follow-up research either encompass implementing the remaining requirements or rolling out the program in schools to see how it performs. The latter will verify whether fulfillment of the requirements gives us an adequate program or if the requirements prove to be insufficient. We believe it's best to stick with Roboter Simulator since it gives us a strong foundation to work with. The advantages are that it runs on Windows, is integrated with BricxCC and has a 3D environment. Another alternative is to build a program from scratch, however this requires a lot more effort for things Roboter Simulator already has. Building on any of the other programs is suboptimal because of the difficulties presented by their respective platforms.

Bibliography

[1] S.H. Choi, A.M.M. Chan, *A virtual prototyping system for rapid product development*, 1994, Computer-Aided Design Volume 36, Issue 5, April 2004, Pages 401-412.

[2] Vijay K. Madisetti, Thomas W. Egolf, *Virtual prototyping of Embedded Microcontroller-Based DSP Systems*, Georgia Institute of Technology, 1995, Journal IEEE Micro archive Volume 15 Issue 5, October 1995, Page 9-21

[3] Johan Schuurbiers, Rachel Crane, Jozef Hooman, Lotte van de Ree *Robotica NLT-module*, Radboud Universiteit Nijmegen, 2013

[4] Robot Virtual Worlds: http://www.robotvirtualworlds.com/

[5] Roboter Simulator: schuelerlabor.informatik.rwth-aachen.de/roboter-simulator

[6] Aegidius Pluess' RobotSim: http://www.aplu.ch/home/apluhomex.jsp?site=75

[7] Daniele Benedettelli's NXCsimulator: http://www.thenxtstep.com/2012/03/nxc-simulator.html

[8] Gambas3: http://gambas.sourceforge.net

[9] Virtual Robotics Toolkit: https://www.virtualroboticstoolkit.com/

[10] LeJOS Website: http://www.lejos.org/

[11] Kammer T., Brauner P., Leonhardt T., Schroeder U. *Simulating LEGO Mindstorms Robots to Facilitate Teaching Computer Programming to School Students.* In: Kloos C.D., Gillet D., Crespo García R.M., Wild F., Wolpers M. (eds) Towards Ubiquitous Learning. EC-TEL 2011. Lecture Notes in Computer Science, vol 6964. Springer, Berlin, Heidelberg pp 196-209 (2011)

Appendix A

Appendix

A.1 Variant A

```
int pwr = 50;

task main(){

    long t1 = CurrentTick();
    SetSensorTouch(S1);
    while(!SensorScaled(S4)){
             SetSensorTouch(S4);
             OnFwd(OUT_BC,pwr);
    }
    NumOut(60,LCD_LINE1,CurrentTick()-t1);
    Off(OUT_BC);
    Wait(10000);
}
```

A.2 Variant B

```
int pwr = 50;

task main(){
    SetSensorTouch(S1);
    OnFwd(OUT_C,pwr);
    Wait(140);
    while(!SensorScaled(S4)){
        SetSensorTouch(S4);
        OnFwd(OUT_BC,pwr);
    }
}
```

Appendix B

Appendix

B.1 Variant A

```
int pwr = 50;

task main()
{
    long t1 = CurrentTick();
    RotateMotor(OUT_B,pwr,9000);
    RotateMotor(OUT_C,pwr,-9000);
    NumOut(60,LCD_LINE1,CurrentTick()-t1);
    Off(OUT_BC);
    Wait(10000);
}
```

B.2 Variant B

```
int pwr = 50;

task main()
{
    RotateMotor(OUT_B,pwr,9000);
    RotateMotor(OUT_C,pwr,-9000);
    Off(OUT_BC);
}
```

Appendix C

Appendix

C.1 Variant A

```
int pwr = 50;

task main()
{
    long t1 = CurrentTick();
    RotateMotor(OUT_B,pwr,-9000);
    RotateMotor(OUT_C,pwr,9000);
    NumOut(60,LCD_LINE1,CurrentTick()-t1);
    Off(OUT_BC);
    Wait(10000);
}
```

C.2 Variant B

```
int pwr = 50;

task main()
{
    RotateMotor(OUT_B,pwr,-9000);
    RotateMotor(OUT_C,pwr,9000);
    Off(OUT_BC);
}
```