Simulation for

LEGO Mindstorms Robotics

A thesis
submitted in partial fulfilment
of the requirements for the Degree of
Master of Software and Information Technology
at
Lincoln University

by

Yuan Tian

Lincoln University

December 2007
Simulation for LEGO Mindstorms Robotics

By Yuan Tian

The LEGO MINDSTORMSTM toolkit can be used to help students learn basic programming and engineering concepts. Software that is widely used with LEGO MINDSTORMS is ROBOLAB™, developed by Professor Chris Rogers from Tufts University, Boston, United States. It has been adopted in about 10,000 schools in the United States and other countries. It is used to program LEGO MINDSTORMS robotics in its icon-based programming environment. However, this software does not provide debug features for LEGO MINDSTORMS programs. Users cannot test the program before downloading it into LEGO robotics hardware.

In this project, we develop a simulator for LEGO MINDSTORMS to simulate the motions of LEGO robotics in a virtual 3D environment. We use ODE (Open Dynamic Engine) and OpenGL, combined with ROBOLAB. The simulator allows users to test their ROBOLAB program before downloading it into the LEGO MINDSTORMS hardware.

For users who do not have the hardware, they may use the simulator to learn ROBOLAB programming skills which may be tested and debugged using the simulator. The simulator can track and display program execution as the simulation runs. This helps users to learn and understand basic robotics programming concepts.
An introduction to the overall structure and architecture of the simulator is given and is followed by a detailed description of each component in the system. This presents the techniques that are used to implement each feature of the simulator. The discussions based on several test results are then given. This leads to the conclusion that the simulator is able to accurately represent the actions of robots under certain assumptions and conditions.

**Keywords:** Graphics, Simulation, LEGO MINDSTORMS, ROBOLAB, OpenGL®, ODE®
Acknowledgements

I would like to take this opportunity to be grateful for the support given to me by my supervisor Keith Unsworth and associate supervisor Alan McKinnon. Without their help and contributions, this thesis work would not be possible. Many thanks go to Walt Abell for his help and advice in this project, and Mark Anderson for his technical support.

I would also like to thank Professor Chris Rogers at Tufts University in the United States for helping to test the simulator. His advice will prove valuable in future enhancements. I also wish to acknowledge the assistance provided by the staff from Science Alive!, Christchurch, New Zealand, and Josh Campbell, teacher from Hagley College, Christchurch, New Zealand in testing the simulator. Thanks for their feedback.

Many thanks also go to the administrative staff and my fellow post-graduates of the Software and Information Technology group at Lincoln University who gave me assistance.

Finally, special thanks go to my wife Hongyu Hu and my parents for their love and support during my post-graduate education.
# Table of Contents

Abstract................................................................................................................................. I  
Acknowledgements ............................................................................................................... III 
Table of Contents................................................................................................................ IV  
The list of Tables.................................................................................................................... VII  
The list of Figures................................................................................................................... VIII  

Chapter 1  
Introduction ......................................................................................................................... 1  
1.1 Background.................................................................................................................. 1  
1.1.1 LEGO MINDSTORMS......................................................................................... 2  
1.1.2 ROBOLAB program ........................................................................................... 3  
1.2 Motivation.................................................................................................................... 6  
1.3 Purpose of the study................................................................................................. 8  
1.4 Outline of thesis....................................................................................................... 9  

Chapter 2  
Literature Review.............................................................................................................. 10  
2.1 Robotic simulation review......................................................................................... 10  
2.1.1 Industrial Robot Simulation ............................................................................. 11  
2.1.2 Educational Robot Simulation.......................................................................... 14  
2.1.2.1 Webots™..................................................................................................... 14  
2.1.2.2 SimRobot.................................................................................................... 16  
2.1.2.3 Summary..................................................................................................... 18  
2.1.3 Existing LEGO MINDSTORMS simulators................................................... 19  
2.1.3.1 VRML based simulator for LEGO MINDSTORMS............................... 19  
2.1.3.2 RoboXAP.................................................................................................... 20  
2.1.3.3 RobertaSim®............................................................................................. 23  
2.1.3.4 Summary.................................................................................................... 24  
2.2 Physics simulation..................................................................................................... 24  
2.2.1 Rigid body simulation....................................................................................... 25  
2.2.2 ODE and OpenGL............................................................................................ 26  
2.3 Summary................................................................................................................... 27  
2.4 Objectives of this study........................................................................................ 28  

Chapter 3  
Implementation..................................................................................................................... 29  
3.1 System Overview....................................................................................................... 29  
3.1.1 Simulation structure......................................................................................... 30  
3.2 System architecture................................................................................................. 31  
3.2.1 TCP/IP communication..................................................................................... 31  
3.2.2 Pre-defined Models using ODE....................................................................... 32
The list of Tables

Table 3.1 LASM instructions of the example program in Figure 1.4 .........................34
Table 3.2 Example of LASM loop structures ................................................................34
Table 3.3 Example of LASM code from a line follower program ..............................35
Table 3.4 For loop structure .......................................................................................36
Table 3.5 The array list stores all LASM program instructions .................................49
Table 3.6 The sorted list ..............................................................................................49
Table 3.7 Table of power measurement .....................................................................51
Table 3.8 Parts of the LASM instructions for collision detection ............................55
Table 3.9 Parts of the LASM instructions for line follower .......................................59
Table 3.10 LASM instructions for “Wait for Darker” and “Wait for Lighter” ..........60
Table 4.1 Comparison of the distance travelled in 2 seconds ..................................77
Table 4.2 Comparison of the angle turned in 2 seconds ..........................................78
Table 4.3 The corresponding LASM “chkl” instruction .............................................82
Table 5.1 The green colour representation with different lightness value ...............94
Table A.1 The sources used in LURobotSim ............................................................99
Table F.1 Specification for each interactive control using mouse and keyboard ......119
Table F.2 Control buttons and explanations ............................................................120
The list of Figures

Figure 1.1 Robotic Command Explorer (RCX) ............................................................. 2

Figure 1.2 Sensors, motors and lamp connect to RCX [15] ......................................... 3

Figure 1.3 A USB IR Tower ..................................................................................... 3

Figure 1.4 Example of icon based ROBOLAB program ......................................... 4

Figure 1.5 Pre-wired ROBOLAB program in Pilot level one .................................... 5

Figure 1.6 An example of an Inventor level ROBOLAB program ......................... 6

Figure 1.7 An example of a line follower from RobertaSim [24] ............................. 7

Figure 1.8 Example of ROBOLAB program for a line following robot ............... 7

Figure 2.1 Industrial robots (KUKA) welding vehicle bodies [33] .......................... 11

Figure 2.2 An example of a resulting simulation from EASY-ROB ....................... 12

Figure 2.3 A screenshot of an EASY-ROB interface ............................................... 13

Figure 2.4 Simulated Sony AIBO vs. Real AIBO .................................................... 14

Figure 2.5 Webots: Scene Tree ............................................................................... 15

Figure 2.6 The SimRobot 3D representation ............................................................ 16

Figure 2.7 Example of screenshot of the SimRobot interface .................................. 18

Figure 2.8 A screenshot of Brower’s simulator interface ......................................... 20

Figure 2.9 Overview of the RoboXAP system structure [3] ..................................... 21

Figure 2.10 ROBOLAB program ............................................................................ 22

Figure 2.11 RoboXAP 2D display ........................................................................... 22

Figure 2.12 Example of screenshot of line follower with RobertaSim ................... 23

Figure 2.13 Example of Hinge-2 joint from ODE user guide [17] ............................ 26

Figure 3.1 Simulation loop logical structure ......................................................... 30
Figure 3.2 The main components in the system ..........................................................31
Figure 3.3 TCP/IP configuration text file for ROBOLAB ..........................................38
Figure 3.4 The Select COM Port list .........................................................................39
Figure 3.5 The relationship between the virtual world and the collision space in ODE ..............................................................................................................................41
Figure 3.6 Example of ODE functions for defining the robot with three wheels .......42
Figure 3.7 A contact point between the wheel and ground........................................43
Figure 3.8 The robot with three wheels model ............................................................45
Figure 3.9 Example of defining four sides of the wall ................................................47
Figure 3.10 The virtual machine structure .................................................................48
Figure 3.11 Example code for collision detection ..................................................54
Figure 3.12 Example of collision detection program..............................................55
Figure 3.13 Example of mapping between world and texture coordinates............57
Figure 3.14 Example of line follower program ......................................................58
Figure 3.15 Matrices in ODE and OpenGL .............................................................61
Figure 3.16 The conversion from the ODE rotation matrix to the OpenGL matrix ....62
Figure 3.17 Example of a rendering function using ODE data...............................62
Figure 3.18 Example of ROBOLAB highlighting feature ......................................63
Figure 3.19 The program structure for rendering program execution trace ..........65
Figure 3.20 A 2D object facing the camera ............................................................66
Figure 3.21 The modified modelview matrix for billboarding ...............................66
Figure 3.22 Example of bill boarding displays .....................................................67
Figure 3.23 Example of MDI windows and its components ..................................68
Figure 3.24 The general structure of an MDI window application.......................68

Figure F.1 Sym text file .............................................................................................114
Figure F.2 Example of ROBOLAB program.............................................................115
Figure F.3 The Select COM Port list .........................................................................115
Figure F.4 A successful connection between ROBOLAB and LURobotSim ..........115
Figure F.5 Confirm or Cancel a model selection........................................................116
Figure F.6 Robot model M1.......................................................................................117
Figure F.7 Robot model M2.......................................................................................117
Figure F.8 Robot model M3.......................................................................................117
Figure F.9 Robot model M4.......................................................................................117
Figure F.10 Robot model M5.....................................................................................117
Figure F.11 Environment model E1...........................................................................118
Figure F.12 Environment model E2...........................................................................118
Figure F.13 Environment model E3...........................................................................118
Figure F.14 The world coordinates for 3D display ..................................................119
Chapter 1  Introduction

1.1 Background

LEGO MINDSTORMS™ and ROBOLAB™ may be used to help students learn science, technology, engineering, and mathematics concepts with hands-on, naturally motivating building sets, programming software, and curriculum relevant activity materials [5][23]. Students can build their robots relatively easily with LEGO® components while concentrating on the main concepts such as science, engineering and programming, which they can learn from the LEGO MINDSTORMS toolkit. Programming software called ROBOLAB may be used to program the actions of the robot. Students download the program into their robot to test its actions. The LEGO MINDSTORMS toolkit is available worldwide and is used in many schools in many countries for teaching purposes.

Simulation can play an important role in education and training [11] as well as in robotics development [9][12]. Currently, in the United Kingdom, schools are required to provide on-line access to a Virtual Learning Environment (VLE) including home access. The e-learning foundation seeks to provide a home-based VLE for school children [53] with simulation playing a fundamental role in this aspect of children’s education.

Simulation not only provides a cheaper, faster, and safer way to design and develop robots, but also enables users to explore an environment, and experiment with robotic events that may be unavailable because of distance, time, or safety factors [4]. The combination of simulation and LEGO MINDSTORMS can provide users with an opportunity to test a robot program as they learn programming and engineering knowledge.

In this project, we develop a simulator for LEGO MINDSTORMS to display the motions of LEGO robots programmed using ROBOLAB, including the capability to
test and debug the ROBOLAB control program. As a result, users who do not have access to LEGO MINDSTORMS are able to learn basic programming and engineering concepts. In addition, users who already have the system can extend their knowledge gained from the LEGO MINDSTORMS system by learning how different aspects of the environment can affect robot behaviour. The following sections will separately introduce the toolkits in more detail.

1.1.1 LEGO MINDSTORMS

LEGO MINDSTORMS toolkits contain a programmable LEGO brick called a Robotic Command Explorer (RCX) (See Figure 1.1), LEGO pieces, sensors and motors that are used to build LEGO robots together with programming software such as ROBOLAB. In addition, there is other programming software available including pbForth and NQC (Not Quite C) mentioned in [20]. Both of them can work with LEGO MINDSTORMS as alternative programming software for the RCX. These programs can be downloaded into the RCX to execute the corresponding robotic behaviour. In this project, we consider ROBOLAB as the programming software for LEGO MINDSTORMS robotics.

![Figure 1.1 Robotic Command Explorer (RCX)](image)

The RCX is the main component of LEGO MINDSTORMS. It controls the output to the motors of a LEGO robot, which may also use inputs from sensors, to achieve programmed behaviour (See Figure 1.2). There are three output ports and three input
ports on each RCX. Figure 1.2 shows an RCX with three types of sensors connected to three input ports, and two motors and one lamp attached to three output ports.

![Figure 1.2 Sensors, motors and lamp connect to RCX [15]](image)

1.1.2 ROBOLAB program

This study uses ROBOLAB as the software to program a LEGO robot. After the program is written, it is downloaded to the RCX via a USB Infrared (IR) Tower (See Figure 1.3).

![Figure 1.3 A USB IR Tower](image)
ROBOLAB is a software package developed at Tufts University and is used with LEGO MINDSTORMS to introduce computer-programming concepts to school age students [5]. The ROBOLAB environment uses LabView visual programming terminology [8], in which programmers wire together images and icons, rather than typing in lines of code. As an example, Figure 1.4 shows a ROBOLAB program that instructs a robot motor to move forward for 2 seconds and then stop. Each icon in Figure 1.4 represents, in LabView Terminology, a Virtual Instrument (VI). This may be constructed from other virtual instruments, resulting in a hierarchy of programming layers. Hence, with ROBOLAB, users including young children can design and create programs that control their LEGO robot by selecting recognizable images [14].

![Figure 1.4 Example of icon based ROBOLAB program](image)

ROBOLAB can share and transfer data between two users via a TCP/IP port. As a result, a ROBOLAB user can easily share his or her program with another ROBOLAB user through a TCP/IP interface. This feature may also allow ROBOLAB to send data to a virtual robot model that can simulate the behaviour of a real LEGO robot. More information about this feature is discussed in chapter 3.

ROBOLAB has three levels of programming. We concentrate on two of these in this project, the Pilot and Inventor levels. The other is the Investigator level [20]. All programs at the Pilot level are pre-wired. At this level a programmer can only change a program’s functionality in a very limited way. School children are likely to use this level to learn basic programming concepts. For example, in Figure 1.5, users can only change the first VI (motor A backward) to “forward” or “turn on lamp”, and the second one (wait for four seconds) to “wait for two seconds” or wait for another defined time period. Pilot Mode does not include any program control structures, such
as loop or conditional statements. Students transfer the program to its destination (e.g. RCX, or other users) by clicking on the white arrow in Figure 1.5.

Figure 1.5 Pre-wired ROBOLAB program in Pilot level one

Compared with the Pilot level, the Inventor level includes a more extensive palette of virtual instruments for students to build their own programs (see Figure 1.6). These provide more flexibility to develop more complex programs. A student can build his or her own program that is displayed within the block diagram window, choosing from the virtual instruments provided in the Function Palette, and wiring them in the order that they will execute. Also, Inventor level allows students to access conditionals and other advanced control structures that may be necessary for developing a more complex program. For example, a student may write a program to cause a robot to run Motor A backward if the touch sensor on port 1 is released (not pressed) and run Motor A forward for 2 seconds if the touch sensor is pressed (see Figure 1.6). As shown in Figure 1.6, the conditional control structure VI checks the touch sensor on port 1. If the touch sensor is released, the program will follow the top stream so that the robot will run Motor A backward. Otherwise it will follow the bottom stream to run Motor A forward for two seconds. There are also many other examples in [20].
1.2 Motivation

As mentioned in earlier sections, LEGO MINDSTORMS toolkits are helpful for teaching students programming and engineering concepts. However, some students may not have access to LEGO MINDSTORMS because of lack of resources. The LEGO MINDSTORMS kit is independent of ROBOLAB software. Therefore, they cannot download their ROBOLAB program to a robot to view the motions. Hence a simulator will allow exploration of programming logic without the need for LEGO hardware at all.

In addition, LEGO MINDSTORMS toolkits have some limitations. Users cannot test, and debug, a ROBOLAB control program before downloading it into a real robot. As a result, a LEGO robot might not have the expected behaviour programmed using ROBOLAB because of errors or bad logic within the program. For instance, users may be dealing with a complex task such as implementing a line following robot using a light sensor. See Figure 1.7. This shows a simulated Lego robot that follows...
an existing path using a light sensor. Figure 1.8 presents a ROBOLAB program for implementing the *line follower robot*. If the sensor detects the (dark) path, the robot will stop motor A and turn on motor C so that it turns and keeps driving forward until the sensor detects the white ground. This means that the sensor has left the path. At this moment the robot stops motor C and turns on motor A so that it drives back to the path. The program keeps repeating the above process to make the robot follow the path.

![Diagram of a line follower robot](image)

**Figure 1.7** An example of a line follower from RobertaSim [24]

![Diagram of ROBOLAB program](image)

**Figure 1.8** Example of ROBOLAB program for a *line following robot*

However, users cannot check if the logic of their ROBOLAB program is correct before transferring it into the RCX. If they use a wrong sensor VI for checking light variation, or wire the icons (VI) in an incorrect order, their robot will fail to follow the
In this case, simulating the action of a LEGO robot will help users to check errors and bad logic in a ROBOLAB program independently of the robot itself.

Additionally, physical constraints and other issues in the environment, in which the robot will run, might cause unexpected behaviour of a robot. In the case of the line follower, it may be hard to set up the light threshold value, because the surrounding environment of the robot may influence the light sensor. For example, the effects of natural light might result in the robot failing to follow the line. Although, simulation still cannot solve the whole problem, it may help users to determine whether the failure of a robot is due to logical programming errors, or environmental problems. For instance, in the line follower case, the program may run correctly in a simulated environment but not the real-life environment. This would indicate that the basic logic of the program is correct, and the problem is caused by the real-life environment, including the possibility of incorrect robot construction.

Also, a student cannot explicitly see execution of the program while the robot runs. In a simulation, this could be done by highlighting each icon as the program executes. This feature would be useful, not only for debugging the control program from the programmer’s point of view, but also for understanding the robotic programming concepts from an educational viewpoint.

1.3 Purpose of the study

In this project, we extend the contribution of LEGO MINDSTORMS in education by developing a simulation tool. In conjunction with ROBOLAB, the simulator should display the action of the LEGO robot in a three-dimensional environment, and provide users with the opportunity to efficiently learn basic programming and engineering concepts without the need for the LEGO MINDSTORMS hardware. For users who are able to access LEGO MINDSTORMS hardware, the simulator should extend their knowledge learned from the toolkits. For example, users should be able to compare results that are output by the simulator with the actual LEGO hardware. Therefore, they should be able to not only test and debug their programs with different
environments, but also learn the problem-solving skills when a simulation result is different from the LEGO MINDSTORMS result. In addition, the simulator should offer users an option to display program execution as the simulation runs. This feature may be useful when users want to understand what each icon (code) does. Also it should make it easier to debug a program that has logical errors.

1.4 Outline of thesis

The following chapters present and discuss the project in more detail including a review of existing robot simulators in the literature and discussion of related work in chapter 2, detailed implementation of the simulator in chapter 3, discussion of some issues based on users’ feedback in chapter 4, and a summary of the entire project and suggestion for future enhancement in the final chapter.
Chapter 2  Literature Review

Computer simulations can be used to represent real-life situations. They allow users to study or try experiments that may be difficult or impossible in real life. Simulations may be useful when a real-life process is too dangerous, takes too long, is too fast, or is too expensive to create. For example, a space flight simulator called Orbiter [28] allows users to explore the solar system using different spacecraft. It is a safe and realistic tool for training and spacecraft development. Microsoft’s Flight Simulator X [35] allows a user to experience the actions of a pilot in a realistic, cheap and safe way. In terms of robot development, simulation can also be a useful tool for developing large industrial manufacturing robots for welding and assembly such as KUKA \(^1\) Robots [29], or small educational robots such as LEGO MINDSTORMS robotics. This chapter discusses the use of robotic simulation in both industry and education, based on several existing robot simulators, although the main focus is on educational robotic simulators that are particularly relevant to this project.

2.1 Robotic simulation review

Robots may be complex, expensive and highly technological systems. Working with robots often requires a good understanding of science, engineering and programming. A simulation allows the robot developer to evaluate different alternatives during the design stage of a robot system, thereby leading to better decisions and cost savings. On the one hand, simulation provides the opportunity to debug and test a robotic program by simulating the output after execution of the program. On the other hand, robotic simulation is also good for developing logical and technical skills.

Several robotic simulators have been developed for different purposes and in different areas including education, computer aided design, and space exploration. Industrial

\(^1\) KUKA company was founded in 1898 in Augsburg, Germany as Keller und Knappich Augsburg.
robot simulators, such as WORKSPACE™ [13] and EASY-ROB™ [31], enable the evaluation of alternatives during the design stage of robot systems within a manufacturing environment. Webots™ [12] and SimRobot [9] are robot simulators that can assist researchers and developers to model, program and simulate mobile robots such as Sony AIBO™ [39] and Khepera™ [32]. The simulator, RoboXAP [3] focuses on agent-based robotics that may be used in the school curriculum as an educational tool.

2.1.1 Industrial Robot Simulation

The field of industrial robotics may be defined as the design and use of robot systems for manufacturing including assembly, welding, painting, and packaging. Figure 2.1 shows an example of an assembly application using KUKA robotics [29]. The simulation of industrial robots often plays an important role in assisting developers to design and plan the robot work cell\(^2\) in a manufacturing environment. An example of a robot work cell is a robotic arm, such as the KUKA robot work cell shown in Figure 2.1.

\[\text{Figure 2.1 Industrial robots (KUKA) welding vehicle bodies [33]}\]

\(^2\) A work cell is an arrangement of resources in a manufacturing environment to improve the quality, speed and cost of a process. In this case, an industrial robot is the resource.
In addition, industrial simulators can provide an opportunity to test the programs for the robot work cells in order to correct their intended paths, and avoid potential collisions between them and plant fixtures. In terms of commercial usability and cost saving, this feature is important during robot system development in manufacturing because of the potential crowded environment of the manufacturing assembly line (see Figure 2.1).

EASY-ROB [31] is an industrial robot simulator that enables developers to plan manufacturing plants that include robot work cells, in a 3D environment. Through industrial robotic simulation software, users may quickly test automation ideas and cost-efficient feasibility studies, and thereby increase reliability and reduce commissioning time [31]. Figure 2.2 presents a simple example of the usability of the simulator. The left robot picks up the box from the palette and moves it to the conveyor. The right robot picks up the box to place it on the second palette after the box has moved along to the end of the conveyor. Developers can test a program for the KUKA robotic arm movement sequences and handling processes through a simulation. In addition, the simulation assists developers to locate the ideal positions for the two robots relevant to the fixtures, i.e. the conveyor and the two palettes, in order to avoid potential collisions between them.

![Figure 2.2 An example of a resulting simulation from EASY-ROB](image)
The free demonstration version of EASY-ROB allows users to import pre-defined 3D robot models and environments described in VRML (Virtual Reality Modelling Language) [27] via a project library. Each file in the library describes one or more 3D robot models and environments. The movements (or paths) of the robotic arm are also pre-programmed using ERPL (Easy-Rob Program Language) and ERCL (Easy-Rob Command Language) [44] and are stored in a file. Figure 2.3 shows a pre-defined KUKA robot whose data is stored in a VRML file. The environment may include fixtures such as a palette and conveyor (see Figure 2.2). EASY-ROB simulates collision detection using Hierarchical Collision Detection. More information about this technique may be found in [21].

The EASY-ROB simulator user interface allows users to control the simulation time using control buttons in the tool bar. Also users can change the viewpoint with the mouse and view different display effects using control buttons (see Figure 2.3). For example, users can click a button to view a wire frame display of a robot instead of a solid one.

![Figure 2.3 A screenshot of an EASY-ROB interface](image)
2.1.2 Educational Robot Simulation

Compared with industrial robot simulations, educational robot simulations focus more on research and teaching. Another distinction is that most educational robots are mobile robots that move within an environment, whereas industrial robots are likely to be fixed in one physical location. This study concentrates on educational robot simulators and the following sub-sections discuss several educational robot simulators namely Webots, SimRobot, RoboXAP, RobertaSim and Brower’s simulator. Note that there are also other educational robot simulators, such as Microsoft Robotics Studio developed by Microsoft. The first version of the simulator was released in December 2006, which is after this study started. For more details of this simulator, the reader can refer to [49].

2.1.2.1 Webots™

Webots [12] is a commercial robot simulator primarily used in universities and research centres for research purposes, such as the field of artificial intelligence and robot control and design. It has been developed for modelling, programming and simulating mobile robots. It provides several pre-defined 3D robot models including Sony AIBO [39], and Khepera [32]. The simulated 3D virtual robots are visually realistic. For example, Figure 2.4 shows a simulated Sony AIBO that looks very similar to the real robot.

![Simulation time control](image1)

![A real Sony AIBO](image2)

![Simulated Sony AIBO and environment](image3)

**Figure 2.4** Simulated Sony AIBO vs. Real AIBO
Webots allows users to import other pre-defined 3D models described using the VRML97 [27] standard. These are stored in the scene tree editor as shown in Figure 2.5. The nodes in the scene tree are the starting point in order to manage objects and properties that describe the robot and its environment in 3D. For example, the environment may be described separately with several nodes including background, light and world information. There may be sub-nodes for specification of the properties in each node. For instance, the node for implementing wheels may have sub-nodes for properties including translation, rotation and scaling (See Figure 2.5).

Figure 2.5 Webots: Scene Tree

To achieve realistic physics in the simulation, Webots uses ODE (Open Dynamic Engine) [17]. For each component of a robot, users can use simple geometric primitives, such as a box, cylinder, or sphere, as a bounding object used for collision detection. In addition, sensors can be attached to a robot model to perform a full range of reactions to the environment and other robots.

Webots allows users to interact with the simulation while it is running. For instance, users can easily change the viewpoint position and orientation using the mouse. Also,
users are able to control the simulation time (pause or speed up) using a button in the tool bar (see Figure 2.4). This feature is particularly useful when users use fast simulation mode and wish to view more simulation detail. Programming the actions of a robot is implemented using the C interface. The program controller may then be transferred to a real robot.

Note however that the version of Webots discussed above does not support ROBOLAB as the program controller.

### 2.1.2.2 SimRobot

SimRobot [9] is a robot simulator that is able to simulate user-defined robots in 3D. It is developed for testing and debugging programs for mobile robots. This may provide benefits for those robots that do not have debugging facilities, such as the Sony AIBO. For example, SimRobot was used in the RoboCup$^3$ competition in 2005. It was used to test and debug programs for the Sony AIBO [39], and simulate a robot and the environment used in the Sony Four-legged League [42], which is one of the RoboCup soccer leagues.

![The SimRobot 3D representation](image)

a) The graphical representation    b) The physical representation

**Figure 2.6** The SimRobot 3D representation

---

$^3$ RoboCup (Robot Soccer World Cup) is an international robotics competition founded in 1993. The aim is to develop autonomous robots with the intention of promoting research and education in the field of artificial intelligence.
SimRobot provides a user-defined robot model that is based on rigid body dynamics using ODE, as for Webots. It allows users to build accurate robot models with a variety of different generic bodies, sensors and actuators that have been implemented using ODE for physics simulation and OpenGL® [25] for graphic modelling. For example, Figure 2.6(a) shows the appearance of the Sony AIBO [39] model with details such as toes, and LEDs on the head, while Figure 2.6(b) shows simple objects such as a box, sphere and capped cylinder.

The physics simulation has been used to improve the existing model of the Sony AIBO for the Sony Four-legged League with regard to both execution of the desired motions and the accuracy of collisions with other robots or other objects. The earlier model does not have collision handling for correct body positions. Thus a ball has to be moved manually when the robot “kicks” it.

The user-defined robot and its environment are specified using an XML-based language called RoSiML (Robot Simulation Mark-up Language) [7]. It is not limited to any special pre-defined mobile robots whereas Webots is. Hence, users are free to design and simulate their own robot model. To generate the 3D presentation, SimRobot uses an object tree (similar to Webots) as shown in Figure 2.7. The object tree is the starting point for specifying all 3D objects and environment information in the scene.

The user interface of SimRobot allows users to handle the simulation within different windows (see Figure 2.7). All objects in the overview window are defined in the object tree. Moreover, SimRobot also implements a set of generic sensors. As a result, a 3D virtual robot can interact with its environment or even another robot with different kinds of sensors such as a touch sensor or distance sensor.

The literature for SimRobot does not mention that ROBOLAB can be used for programming the robot.
2.1.2.3 Summary

Webots and SimRobot both use the rigid body dynamics physics simulation software called ODE, combined with a graphics library to try and make the simulation as realistic and accurate as possible. In addition, they both use a scene tree to create and manage the 3D display. The distinction is that Webots allows users to adjust the simulation time whereas SimRobot does not, and SimRobot allows users to define their own robots whereas Webots does not. Based on the studies of these two mobile robot simulators, ODE combined with OpenGL are good physics and display simulation libraries for robot simulation. The later sections will introduce rigid body dynamics and ODE in more detail.

Figure 2.7 Example of screenshot of the SimRobot interface
However, for our purposes, a disadvantage of the above two simulators is that they are not able to connect to ROBOLAB. The following section presents three educational simulators that are able to connect to ROBOLAB.

2.1.3 Existing LEGO MINDSTORMS simulators

As mentioned in chapter one, LEGO MINDSTORMS toolkits have been used for teaching science and engineering concepts in many schools around the world. Simulators have been developed to combine with the toolkits in order to extend the contribution of LEGO MINDSTORMS as a useful teaching tool. They are often used to test and debug a robot program before downloading it to the hardware, and also to teach students programming, engineering or other concepts in conjunction with LEGO MINDSTORMS toolkits. The sections below discuss several studies based on LEGO MINDSTORMS and ROBOLAB.

2.1.3.1 VRML based simulator for LEGO MINDSTORMS

Brower and Subramanian present their simulator for LEGO MINDSTORMS in [2]. This simulation supports some basic functionality of LEGO MINDSTORMS such as moving forwards, backwards, and rotating. The simulator uses VRML VI that is integrated into ROBOLAB. As a result, both programming and viewing of a simulation can be done within the ROBOLAB environment. VRML VI supports most of the VRML 2.0 [26] specification to generate a display of the environment and the LEGO robot. However, VRML VI does not support some important features for graphic display including textures and extrusions. In addition, it does not provide any detailed physical interactions such as collisions. It is still in the development stage.

Although the simulated LEGO robot can interact with the environment through sensors (to follow a line for example) the simulation is still limited to a 2D environment and a simple two-motor LEGO vehicle. As shown in Figure 2.8, the display of the robot and the environment are limited by a lack of graphics features and are less realistic than Webots and SimRobot. Figure 2.8 shows the interface of the simulator. The simple 2D object in the middle of the square represents the LEGO
robot. Users are able to view information related to the robot including speed, velocity and location history in a display area at the bottom of the screen.

The simulator does not provide any feature to trace program execution in order for example to indicate where a program error might occur.

Figure 2.8 A screenshot of Brower’s simulator interface

2.1.3.2 RoboXAP

Compared with Brower’s simulator, RoboXAP [3] tries to give students an opportunity to learn four educational topics – “agent-based concepts, programming basics, mechanical engineering and physical world constraints” according to [3], in conjunction with LEGO MINDSTORMS and ROBOLAB. It has been developed primarily to overcome difficulties encountered when implementing educational
robotics curriculum for school children. The difficulties include how to debug a ROBOLAB program and physical affects in the real world, such as noise.

Figure 2.9 Overview of the RoboXAP system structure [3]

RoboXAP translates the ROBOLAB programs into XML (Extensible Makeup Language [45]) codes through an XML-based Agent Protocol Language (XAP) translation engine, and simulates the resulting movements of a predefined LEGO robot in a Flash™ [22] environment. The XAP translation engine acts as an agent between ROBOLAB and the output representation that may be either a simulator or robot hardware (see Figure 2.9). The robot hardware includes two robots, LEGO MINDSTORMS and the Sony AIBO. The translation engine combines XML and a behaviour language called CML (the Cognitive Modelling Language) [6]. CML is designed for behaviour specification in simulation environments. Note however, this feature is hidden from the users. Precisely how the simulator teaches agent-based concepts is not clearly stated in the literature.

The XAP engine cannot fully implement some ROBOLAB program control structures. For example, the XAP engine is not able to implement a “Jump” command, used in CASE and LOOP structures, in order to go back to an earlier address or jump ahead to a later address. These types of control structures are
important to implement complex tasks such as implementing a line follower as mentioned in section 1.2.

In addition, the simulated robot and environment are limited by a 2D display representation. Figure 2.11 presents a simple 2D four-wheeled robot at the left bottom corner. The arrow indicates the directions of the robot movements. The movements of the robot are programmed by ROBOLAB (see Figure 2.10). It instructs the robot to move forward for 4 seconds, rotate for 4.7 seconds, and then move backward for 4 seconds.

![Figure 2.10 ROBOLAB program](image1)

![Figure 2.11 RoboXAP 2D display](image2)

In the preliminary version, the simulator is independent of ROBOLAB in contrast to the embedded nature of Brower’s simulator. Users need to manually launch the simulator in order to receive the commands from ROBOLAB and display the behaviour of a robot. The later version may be automatically launched from the ROBOLAB environment. RoboXAP does not allow users to view program execution.
2.1.3.3 RobertaSim®

The educational robot simulator RobertaSim® [24], developed in Germany, is already used in many schools in that country. It is used with LEGO MINDSTORMS hardware for teaching programming and engineering skills. It is primarily designed and used for the Roberta [46] project that is an on-going educational project for teaching and training robotic concepts using LEGO MINDSTORMS toolkits. The preliminary version does not use ROBOLAB as the programming software. However, a newer version may be used with ROBOLAB via a TCP/IP interface.

Compared to RoboXAP and Brower’s work, RobertaSim uses 3D display representations with physics simulation so that the virtual robot is more realistic and is able to perform more complex tasks such as a line follower (see Figure 2.12). Users can download a ROBOLAB program into the simulator using a TCP/IP interface. However, for version 1.0 tested in this study, users can neither view the program execution nor control the simulation time. In addition, the simulator is not able to change motor power levels. As a result, for example, a user is not able to view the difference two motor power levels has on the speed robot travels. Note that a later version has resolved this issue [16].

Figure 2.12 Example of screenshot of line follower with RobertaSim
2.1.3.4 Summary

The educational simulators described above have the common feature that they can all connect to ROBOLAB for programming a robot’s action. However, although they can implement most of a robot’s functionality, the representation of the behaviours is limited: Brower’s simulator and RoboXAP are both limited by the 2D display representation, while version 1.0 of RobertaSim is limited by a lack of functionality in terms of changing parameters such as motor power level. Thus the virtual robot model and environment is less realistic compared to Webots and SimRobot.

In addition, in terms of testing and debugging ROBOLAB programs, users are not able to view the execution of the robot program. This feature would be very useful so that users may locate programming errors more efficiently.

The combination of the physics simulation, highlighting program execution, and ROBOLAB connection is the major focus in this study. More details about physics simulation are presented in the following sections.

2.2 Physics simulation

Simulation of a robot has a lot in common with the simulations required for the development of modern computer games. “A game engine is the core software component of a computer or video game or other interactive application with real-time graphics” [30]. Each game engine must come with a physics engine to carry out the simulation and prediction under different conditions in the simulated world. There are two types of physics engines, real-time and high precision. According to [36], “High precision physics engines require more processing power to calculate very precise physics and are usually used in computer-animated movies”, while real-time physics engines simplify their calculations and lower their accuracy so that they can be performed in time for the game to respond at an appropriate rate for game play. As a result, video games or simulations normally use real-time physics engines, using rigid body dynamics, for faster and easier calculations. There are a number of open
source real-time physics engines available on the World Wide Web including ODE [17], Bullt [37], and OPAL [38]. Game software that uses ODE as a physics engine, include GripShift® for Playstation 3™ and PSP™, Call of Juarez® for PC, and others according to the ODE website [17].

2.2.1 Rigid body simulation

According to [43], “a rigid body is an idealization of a solid body that has a finite size in which deformation is neglected. In other words, the distance between any two given points of a rigid body remains constant in time regardless of external forces exerted on it.” This means a rigid body cannot be deformed at all, whereas a non-rigid body can. In the real world, cars are expected to skid and tumble, brick walls are expected to crumble when struck, and bridges are expected to sway when walked on. Rigid body simulation is a common technique for generating these motions. For an articulated figure, such as a human being, each section of the figure is built as a rigid body, and the flexing of bones under strain or similar effects is ignored. This can simplify the physics equations in order to calculate dynamic behaviour in time. As noted in the previous studies for Webots [12] and SimRobot [9], rigid body simulation is an appropriate method to make a robot simulation realistic and accurate.

To simulate the motion of a rigid body, some properties must be included such as mass, velocity, acceleration, and force. The velocity and acceleration of each body is calculated based on the body’s centre of mass as in the equations shown in [54]. Rigid body simulation libraries often come with many kinds of joint types such as hinges and ball-and-socket joints that can connect bodies together. “In rigid body simulation, each body starts out with six degrees of freedom (three for position and three for orientation)” [18]. Constraints are enforced by the simulator, for example by automatically calculating the constraint forces applied to each body. Constraints often take the form of joints that can connect pairs of bodies together. The solution of the
velocity equations of the simulated system is based on algorithms due to Stewart and Trinkle [19] and Anitescu and Potra [1].

2.2.2 ODE and OpenGL

Open Dynamic Engine (ODE) [17] is an open-source library for simulating articulated rigid body dynamics. It is platform independent and widely used in many robot simulators including Webots and SimRobot. ODE simplifies the physics equations by assuming the centre of mass of a rigid body is located at the origin (0, 0, 0) relative to the body’s geometry position. ODE provides advanced joint types and integrated collision detection with friction. A developer can connect various shapes together to construct a body with different types of joints. There are five types of joints namely Ball-Socket, Hinge, Hinge-2, Slider, and Universal. The details of these joints can be found in [17]. In addition, in order to perform simulation in real-time, the physical shape of a simulated object is often simplified. In the cases of Webots and SimRobot, ODE has been used to implement the physics effects of rigid bodies. Figure 2.6 (b) shows a simple physical representation constructed using a cylinder, sphere and box. ODE is useful for simulating moving vehicles and objects in a virtual reality environment. For example, the Hinge-2 joint is excellent for simulating wheeled vehicles (see Figure 2.13). It can join two bodies through two axes and one anchor. The wheel can rotate around axis 2 and change direction, such as turning left, around axis 1.

![Figure 2.13 Example of Hinge-2 joint from ODE user guide [17]](image-url)
ODE is a computational engine, and is completely independent of any graphics library. As a result, developers need to use a graphics library, such as OpenGL, to make the virtual robot visible to a user. Note that the graphics library is not limited to OpenGL. There are also other graphics libraries available such as Microsoft’s DirectX. However, OpenGL is the graphics library used in this thesis. OpenGL is an open source cross-platform software library for 2D or 3D graphics. It is widely used in CAD, virtual reality, scientific visualization, information visualization, flight simulation and video game development. The OpenGL API (Application Programming Interface) [25] consists of over 150 different functions that can draw complex three-dimensional scenes from simple primitives. The developer can construct a virtual robot by joining bodies (using ODE, for example) that are described using OpenGL. The joints that are created need not be explicitly displayed. More detail about the relationship between ODE and OpenGL is discussed in chapter 3.

2.3 Summary

Referring to section 2.1.2.3 we note that the simulators that incorporate the physics engine (ODE), together with OpenGL, can perform motions of a virtual robot in a virtual environment much more accurately and realistically compared to those simulators that do not use any physics engine. However, the simulators considered in 2.1.2.1 and 2.1.2.2 cannot be combined with ROBOLAB to teach students basic programming and engineering concepts. On the other hand, referring to 2.1.3.4, current educational simulators that can connect to ROBOLAB do not, in general, provide realistic and accurate simulations. They are also restricted to a limited range of types of robots. RobertaSim (version 1.0) is the most complete and successful educational simulator, but it also has a limited range of types of robots. Most importantly none of the existing simulators allow the user to view program execution as the simulation runs.
2.4 Objectives of this study

The overall objective of this project is to develop a LEGO MINDSTORMS simulator, combined with ROBOLAB, which can be used to test and debug a control program, written using ROBOLAB, which simulates the behaviour of LEGO robots using ODE and OpenGL. The simulator will also trace program execution, thereby giving users a tool to help with understanding programming basics. In order to achieve the overall objective, the work is broken down into several steps as follows.

1. Develop a basic LEGO MINDSTORMS simulator using ODE, OpenGL and ROBOLAB, which can display basic movements of a simple pre-designed virtual LEGO robot in a 3D world.

2. Implement a technique for tracing program execution, which can trace the code as the simulator executes to display the code that is being executed within the simulation environment at any moment in time.

3. Develop and implement sensor interaction including those for touch and light sensors, and then test the simulator on predefined robots and environments of increasing complexity.

4. Test the preliminary version to gain feedback from users for further enhancements.

At the initial stage, as a demonstration of a proof of concept, the virtual LEGO pieces will be restricted to simple shapes. All these objectives will be discussed in the detailed implementation chapter that follows.
Chapter 3  Implementation

This chapter presents techniques that are used to develop and implement the simulation system for LEGO MINDSTORMS and ROBOLAB. The simulator is named as **LURobotSim**. The chapter starts with an overview of the system in sections 3.1 and 3.2 in order to show its overall structure. This is followed by a detailed description of each component of **LURobotSim**.

### 3.1 System Overview

**LURobotSim** displays the actions of a pre-defined robot in a 3D environment. ROBOLAB is used to program the actions of the robot. The program is transferred to the simulator as LEGO Assembly Language (LASM) code. **LURobotSim** reads the LASM instructions, and generates the corresponding actions of a virtual LEGO robot in a Windows application using OpenGL together with ODE.

The geometric description of the robot, such as its shape and position, is specified using ODE. OpenGL uses the same geometric descriptions to create displays of three-dimensional scenes. In addition, ODE uses physical properties, such as mass, velocity and friction, to represent the physical objects and simulate how they behave. The detailed implementation of the pre-defined robots, and the physics simulation, is described in later sections where it is also shown how users are able to interact with **LURobotSim**, for example by changing the viewpoint and the simulation timing. In the latter case, users are able to slow the actions of a robot, or make it pause, in order to view its detailed operation and its environment.

Before discussing the implementation in more detail, the overall logical structure of **LURobotSim** is presented in the following section. This structure will give the reader an overall idea of how **LURobotSim** works.
3.1.1 Simulation structure

As it processes the stored LASM instructions, \textit{LURobotSim} simulates and displays the actions of a pre-defined robot using ODE and OpenGL functions in a simulation loop outlined in Figure 3.1. The actions are based on information from the virtual machine (\textit{Controller}), such as the speed and orientation of the robot. The \textit{Controller} is the component that stores and processes the LASM instructions. Section 3.6 discusses this in more detail.

The calls to ODE to carry out a simulation step, and the calls to OpenGL to render the scenes, are completely independent. This means the display of the current state of the robot does not cause ODE to execute a simulation step. Before the simulation loop starts, \textit{LURobotSim} initialises ODE and OpenGL to display the current state of the robot. During each iteration of the simulation loop, the simulator uses OpenGL to display all the VIs that are used in the ROBOLAB program. The Controller is then called to process one LASM instruction in order to provide information for ODE to calculate the next state of the robot. Also, OpenGL is used to highlight the icon of the VI that is currently executing. Once ODE calculates the next state, OpenGL is used to display it. See Figure 3.1.

\begin{figure}[h]
\begin{tabular}{|l|}
\hline
\begin{itemize}
\item Initialise ODE and OpenGL
\item Display current state of the robot using OpenGL
\end{itemize}
\textbf{While} (program not finished) \\
\{ \\
\begin{itemize}
\item Display the icons for all VIs in the ROBOLAB program
\item Call virtual machine, \textit{Controller}, to process one LASM instruction
\item Highlight the icon of the VI that is currently executing
\item Update simulation time
\item Calculate the next state using ODE
\item Display next state of the robot using OpenGL
\end{itemize}
\}
\end{tabular}
\caption{Simulation loop logical structure}
\end{figure}

Appendix D presents the main procedures that are called from the simulation loop (the while loop as shown in Figure 3.1).
3.2 System architecture

To understand the relationship between each component of the system, this section provides a description of the system architecture. The system has two main parts, namely the robot-programming environment (ROBOLAB) and the robotic simulator (LURobotSim) (see Figure 3.2). Within LURobotSim, as the Robot Controller interprets the LASM instructions the user can change the 3D view via a graphical user interface (GUI). This section explains each component in general. Detailed descriptions of the components in Figure 3.2 are given in later sections.

![Figure 3.2 The main components in the system](image)

3.2.1 TCP/IP communication

LURobotSim communicates with ROBOLAB through a TCP/IP port. It acts as a server application and receives and sends data via a TCP/IP socket. On the client side, ROBOLAB sends program data to the simulator through the same TCP/IP port. Section 3.4 explains how the TCP/IP connection between ROBOLAB and the simulator is set up.
3.2.2 Pre-defined Models using ODE

A physics model is defined as many objects (parts) joined together. ODE defines each part of the robot model including its mass, velocity and friction as well as the geometry (cylinder, sphere, and box). For example, a four-wheeled LEGO robot can be built with four cylinders as wheels and one box as a chassis. A hinge joint is applied to attach each wheel to the chassis. Section 3.5 provides a detailed explanation of the physics of a robot using ODE.

3.2.3 Robot controller

The robot controller stores and processes LASM instructions received from ROBOLAB. The actions of the robot are generated by sending data to the relevant ODE functions as the controller executes the stored LASM instructions. Section 3.6 discusses more details of the implementation of the controller and the interpretation of LASM instructions in ODE.

3.2.4 OpenGL rendering of Models

Once ODE has defined the robot, OpenGL displays the bodies such as boxes, spheres and cylinders using the geometrical descriptions provided by ODE, and the environment with the use of lighting and texture mapping. The texture mapping is used to display both the ROBOLAB program icons and the image of the robot. Section 3.7 presents details of how the rendering of the scene is implemented.

3.2.5 GUI design

Users can visualize the behaviour of a 3D virtual robot, and are able to interact with it, within the 3D environment through a GUI. For instance, users can change the viewpoint, using the keyboard or mouse, and adjust the simulation time to make it run slower, or pause, for more detailed viewing. In addition, program execution is tracked and is displayed to the user with ROBOLAB program icons in a display area, or in a
3.3 LEGO Assembly Language (LASM)

Before discussing detailed implementations of each component, this section introduces LASM instructions in more detail. This will give the reader required background information for later sections on the TCP/IP connection and the controller.

As described in chapter 1, a ROBOLAB program comprises a set of Virtual Instruments (see Figure 1.4). When a user transfers a program to an RCX, ROBOLAB will generate instructions in LASM that the RCX understands. An LASM program is a list of instructions. An instruction consists of a command, which may have associated parameters. Most of the instructions have one or more parameters (see for example Table 3.1). The overall structure of a program begins with the command “task” or “sub” and is terminated by the command “endt” or “ends”.

Table 3.1 shows an example of the LASM instructions for the ROBOLAB program in Figure 1.4. The index column indicates the line number in the LASM code. The VI column indicates the corresponding ROBOLAB VI that each group of LASM instructions corresponds to.

For a more complex ROBOLAB program such as the line follower shown in Figure 1.8, a loop control structure needs to be used. In this program, there are two loop structures that are used to repeatedly check the light sensor value until it matches a light threshold value and it continues that checking as the robot travels along the line. The two loop structures are a **While** Loop and a **Forever** Loop, both shown in Table 3.2. The **While** loop is used to check if the light sensor value matches the light threshold, and the **Forever** loop (resulting from the “Jump” VI pair) is used to keep the robot following the line indefinitely.
Table 3.1 LASM instructions of the example program in Figure 1.4

<table>
<thead>
<tr>
<th>Index</th>
<th>Command</th>
<th>Parameter</th>
<th>VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>task</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>tmrz</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>set</td>
<td>24,9,2,1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>set</td>
<td>0,47,2,5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>mulv</td>
<td>47,2,20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>set</td>
<td>5,0,0,47</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>out</td>
<td>2,1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>wait</td>
<td>2,200</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>out</td>
<td>1,1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>endt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2 Example of LASM loop structures

<table>
<thead>
<tr>
<th>Type</th>
<th>Structure</th>
<th>VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>While</td>
<td>startwhilelabel:</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>chkl s1, v1, relop, s2, v2, endwhilelabel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(while code)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jmpl startwhilelabel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>endwhilelabel:</td>
<td></td>
</tr>
<tr>
<td>Forever</td>
<td>startforeverlabel:</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>(loop code)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jmpl startforeverlabel</td>
<td></td>
</tr>
</tbody>
</table>
Table 3.3 shows a sequence of LASM instructions, which is part of the line follower program. This demonstrates how While and Forever loop structures work. The While loop uses the “chkl” command. Referring to Table 3.2, for the “chkl” command, each of s1 and s2 is the source of a number, each of v1 and v2 is the value of the number, and relop is a relational operator used for comparisons. Thus, in Table 3.3, the source number (2) indicates a constant value (50) and the source number (9) indicates sensor (0)\(^4\). The relational operator value (1) indicates “less than”. This While loop repeatedly checks if the threshold value (50) is less than the sensor value on port 1 of the RCX. If the condition evaluates to False, the command will jump to the given address (Label1003), otherwise, the loop will continue execution. The

---

\(^4\) The variable, 0, indicates port 1 of a RCX. It holds the sensor value on port 1.
**Forever** loop structure starts from the beginning of the program and includes the **While** loop structure and the LASM instructions from the other VIs.

In each kind of loop structure, a label such as “Label1002” is used as a reference to a memory address so that a loop can start or end at a certain point in a program. For example, a label combined with the “jmpl” command within a loop, can cause the program to move to the start or end point of the loop. A **For** loop structure also uses the “jmpl” command in conjunction with a label to loop through certain commands a number of times and then jump out of the loop to the end loop label. For example, Table 3.4 shows a **For** loop structure, and the relevant VI. The command “setv” is used to set up the loop variable counter (33) to have a constant value (5) according to the source type indicated by the value (2). The “decvjnl” command decreases the variable counter (33) as it executes, and jumps out of the loop (Label1003) when the variable counter value becomes negative.

<table>
<thead>
<tr>
<th>Structure</th>
<th>VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>setv 33, 2, 2</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Label1002: (startlooplabel)</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
<tr>
<td>decvjnl 33, Label1003 (endlooplabel)</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
<tr>
<td>{loop code}</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
<tr>
<td>jmpl Label1002</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Label1003: (endlooplabel)</td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

In general, there are two kinds of commands in a program: a program command and a direct command. The difference between them is that the direct command needs to return a response to ROBOLAB whereas a program command does not need to respond. When ROBOLAB transfers the direct commands to the RCX, it expects some response from the RCX. For example, ROBOLAB will send a direct command called “poll” to expect an answer from the RCX in order to test if the RCX is available for receiving an LASM program, before ROBOLAB sends the entire program. In Appendix C, the direct commands to carry out communication between
ROBOLAB and \textit{LURobotSim} are described. These direct commands play an important role in implementing the TCP/IP connection feature of \textit{LURobotSim} which provides this communication.

Appendix A lists the LASM instructions, including direct commands, relevant to all the ROBOLAB code used in this thesis. For a listing of all LASM instructions, the reader is referred to [10].

\section*{3.4 TCP/IP connection}

As mentioned in chapter 1, ROBOLAB is able to share a program between two users via a TCP/IP port. This feature may be extended to allow ROBOLAB to communicate with a simulator. Version 2.9 of ROBOLAB with patch, 2.9.3, allows users to use this feature.

This section expands the general concept introduced in 3.2.1 by explaining the implementation of the connection between ROBOLAB and \textit{LURobotSim} in more detail.

Implementation of a TCP/IP application requires two parts, the server side application and the client side application. To achieve communication between the server and the client, the server application creates a listening socket based on a TCP/IP port and waits for a client connection. When a client connects to the listening socket, the server accepts the connection and begins to receive and respond to messages from the client. In the case of communication between ROBOLAB and \textit{LURobotSim}, \textit{LURobotSim} is the server application that waits for a connection request from the client, ROBOLAB, by listening on a specified port. The following two sections explain how each part works.
3.4.1 Server side implementation

For implementation of the server application, \textit{LURobotSim} uses the Windows Socket API via a Winsock library [41]. This creates a listening socket on a specified computer port from 1024 to 5000. Microsoft Visual C++.NET 2003 provides the Winsock library C/C++ header file. Appendix C presents a guide for the configuration of the library in Microsoft Visual Studio.NET 2003. In this project there are six steps to program a Winsock application using the corresponding Winsock functions. These together with some programming details are also described in Appendix C. Note that all Winsock references are given in [41].

3.4.2 Client side implementation

For the client side connection, ROBOLAB needs to specify the port that is used for communication, the IP address of the computer that is being used to execute \textit{LURobotSim} and the server name. If both \textit{LURobotSim} and ROBOLAB execute on the same computer, the IP address should be \textit{localhost}. A text file, namely “sym.txt” is created in the ROBOLAB system directory in order to store and provide the TCP/IP connection information for ROBOLAB.

![sym.txt - Notepad](image)

**Figure 3.3** TCP/IP configuration text file for ROBOLAB

Figure 3.3 shows the necessary text file for configuration of the ROBOLAB connection in the text file, including “SymName” for the server name (LegoSim), “SymAddr” for the IP address of the server side (localhost), and “SymPort” for a TCP/IP port for communication (4266, \textit{LURobotSim} builds a listening socket on this port). After this configuration is specified, when users create a new ROBOLAB
program, they should be able to select the server port in the COM port list as shown in Figure 3.4.

![Figure 3.4 The Select COM Port list](image)

### 3.5 Pre-defined Objects

As briefly introduced in section 3.2.2, ODE plays an important role in the construction of the pre-defined objects, namely the robots and their environments. Also as mentioned in section 3.1.1, *LURobotSim* pre-defines these objects before the simulation starts. The aim of this section is to present the detailed implementation of the pre-defined robot model and the environment in which the virtual robot is placed.

*LURobotSim* currently provides five robot models as below.

- **M1.** One-motor robot without a sensor
- **M2.** One-motor robot with a touch sensor
- **M3.** Robot with three wheels and two motors
- **M4.** Robot with four wheels and two motors
- **M5.** Robot with three wheels, two motors and a light sensor

It also specifies three environment models,

- **E1.** A flat plane
- **E2.** A surrounding wall sitting on the flat plane
- **E3.** Four black lines that form an enclosed square area on a white plane

The user guide in Appendix F shows details of these pre-defined robot models and the three environment models.
3.5.1 Physics description using ODE

To define a virtual robot, a virtual world in which the robot will run needs to be created using ODE. This contains all the rigid bodies and their joints. Note that all objects in the virtual world must be moveable (dynamic) objects. The virtual world is created using the function \texttt{dWorldCreate()}. An object is defined in the virtual world by defining its wire frame.

- **Frame**

  The wire frame of an object consists of mass, position, orientation and dimensions. However, the frame lacks geometry descriptions for an object, such as its bounding surfaces. The lack of these geometry descriptions allow an object to sink into a second object when they collide.

- **Space**

  ODE solves the “collision sinking” problem by using a collision world named “space” that is associated with the virtual world. The geometry of the object is defined in this space.

  Thus, while the robot dynamics are performed in the virtual world, this may lead to a collision. Calculations related to this collision are carried out in the space. This is explained in more detail in 3.6.3.1. Note that if developers implement their own collision detection system, they do not need to define a space at all. \texttt{LURobotSim} uses the built-in collision detection system with the space defined by the function \texttt{dSimpleSpaceCreate}. This is used only for creating the space that defines small numbers of objects. The time required to do intersection testing for \( n \) objects is \( O(n^2) \). Hence, for large numbers of objects this can become time consuming [17]. There are two other functions available to create a space for large numbers of objects, which take less time than \texttt{dSimpleSpaceCreate} by including collision culling\(^5\) (for more detail, see section 10.6 in [17]).

\(^5\) Collision culling will quickly identify which pairs of geoms are potentially intersecting.
Figure 3.5 The relationship between the virtual world and the collision space in ODE

The general structure of a virtual world and space in ODE is shown in Figure 3.5. Each body defined in the virtual world can have only one geometry object associated in collision space (see Body1 and Geometry1 in Figure 3.5). However, in the case of a static (non-moveable) object, it is only defined in the space (see for example, Geometry3 in Figure 3.5), not in the virtual world. Thus, a static object is able to collide with the other objects in the space, but is not able to move in the virtual world. Whereas, an object that is only defined in the virtual world but not in the space (see for example, Body3 in Figure 3.5), is a moveable object, but it is not able to collide with the other bodies (Body1 and Body2).

Figure 3.6 shows a detailed ODE example of a pre-defined robot with three wheels and its collision space. Lines 1 and 2 in Figure 3.6 refer to the creation of the virtual world and the collision space. In lines 5 and 6, LURobotSim defines the Body property for each object in the virtual world (World). In this case, there are nine objects defined for the robot and its environment, namely RCX, four wheels, and four walls. The geometry (Geom) for each object is defined in the collision space (Space) in line 14 (for RCX) and line 19 (for wheel). The geometry uses the same dimensions as the body such as sides for RCX and radius for wheel. Once the two descriptions are combined together using the function dGeomSetBody in lines 15 and 20, the collision engine is able to obtain the position and orientation of the object from the Body, and pass them to the geometry for collision detection during the simulation.
1. World = dWorldCreate(); // create a new dynamics world
2. Space = dSimpleSpaceCreate(0); // create a new collision space
   // if the argument is non-zero, insert the new space into that space
3. dCreatePlane(Space, 0, 1, 0, 0); // create a static plane
4. dWorldSetGravity(World, 0, -9.81, 0); // set up global gravity
5. for (int i = 0; i<8; i++) { // create the bodies in the world, assumed
6.    ObjectG[i].Body = dBodyCreate(World); } // to be 8 of them
7. dReal sides[3] = {0.125, 0.05, 0.064}; // define the RCX dimensions (metre)
8. dReal radius = 0.021; // define wheel radius (metre)
9. dReal length = 0.013; // define wheel width (metre)
10. dReal DENSITY = 1428; // define the density for RCX (kgm⁻³)

   // Set up for RCX
   // set body frame position at global coordinate
11. dBodySetPosition(ObjectG[0].Body, 0, 0.05, 0);
   // set mass parameters (m) to represent a box of the given
density and dimensions
12. dMassSetBox(&m, DENSITY, sides[0], sides[1], sides[2]);
13. dBodySetMass(ObjectG[0].Body, &m); // set the mass for the body frame
   // define geometry shape for collision detection
14. ObjectG[0].Geom[0] = dCreateBox(Space, sides[0], sides[1], sides[2]);
   // Bind the body and geometry together
15. dGeomSetBody(ObjectG[0].Geom[0], ObjectG[0].Body);

   // Set up for front wheel
16. dBodySetPosition(ObjectG[1].Body, 0.09, 0.021, -0.005);
17. dMassSetCappedCylinder(&m, DENSITY * 0.5, 3, radius, length);
18. dBodySetMass(ObjectG[1].Body, &m);
   // define geometry for collision space
19. ObjectG[1].Geom[0] = dCreateCCylinder(Space, radius, length);
20. dGeomSetBody(ObjectG[1].Geom[0], ObjectG[1].Body);

   // Setting up for back wheels is similar to the front wheel except the
   // position.
   // create the hinge joint to connect each wheel to the RCX
21. for (int i = 0; i<=2; i++) {
22.    Joint[i] = dJointCreateHinge (World, jointgroup);
23.    dJointAttach(Joint[i], ObjectG[0].Body, ObjectG[i+1].Body);
24.    dReal *a = dBodyGetPosition(ObjectG[i+1].Body);
25.    dJointSetHingeAnchor(Joint[i], a[0], a[1], a[2]);
26.    dJointSetHingeAxis(Joint[i], 0, 0, 1); }

**Figure 3.6** Example of ODE functions for defining the robot with three wheels
To construct the robot model, different bodies can be joined together using different joints. In this case, the three wheels are connected to the RCX using a hinge joint (see lines 22 to 26). Section 3.5.2 gives more detailed explanations of the hinge joint.

In addition, the world gravity vector (x, y, z) is implemented using the ODE function `dWorldSetGravity(World, 0, -9.81, 0)` in line 4. This informs ODE that the earth’s global gravity vector is in the negative y direction with a value 9.81 \( \text{ms}^{-2} \). ODE does not have built-in units for the simulation. `LURobotSim` uses metres and seconds as the general units for physics simulation. The density of an object has units of \( \text{kgm}^{-3} \).

![Figure 3.7](image)

**Figure 3.7** A contact point between the wheel and ground

When two geometries collide, a contact joint is created based on the two objects. Friction forces in two directions (F-dir 1 and F-dir 2) occur. These are perpendicular to the contact normal. “F-dir 1 is a vector that defines a direction along which frictional force is applied, and F-dir 2 is a vector computed to be perpendicular to F-dir 1”[17] (see Figure 3.7). In this project, the Coulomb friction coefficient [47], \( \mu \) is specified for both of the directions. Rougher surfaces such as rubber in contact with other surfaces tend to have \( \mu \) values between 1 and 2. In ODE, when \( \mu \) becomes zero, the contact is frictionless, whereas the contact never slips when \( \mu \) is \( d\text{Infinity} \). Note that “infinity” (\( d\text{Infinity} \)) friction contact is less time consuming to compute than finite friction contact. As a result, `LURobotSim` uses \( d\text{Infinity} \) to simulate the friction between wheels and the ground in perfect conditions (wheels never slip). The reader can refer to section 7.3.7 of the ODE user guide [17] for more details of \( \mu \) and other
friction parameters used in different situations. The contact joint is fundamental to the collision detection system. Section 3.6.3.1 presents more information about this joint when considering collision detection relevant to the implementation of a touch sensor.

### 3.5.2 Virtual robots

As discussed in section 3.5.1, ODE builds the physics and geometrical description of each part of the robot. There are six types of geometry in the ODE library namely sphere, box, cylinder, ray, triangle mesh and user-defined class. *LURobotSim* uses the box and cylinder for robot modelling. The following paragraphs describe the specification of one of the *LURobotSim* robot models in ODE, namely the robot model that has one front wheel as the driving wheel and two back wheels connected to the RCX (see Figure 3.8). The ODE code for this model is shown in Figure 3.6.

In this case, each wheel is described to ODE as a capped cylinder, and the RCX as a box. In ODE terms, each part of the robot is treated as an individual object (*ObjectG*) (see Figure 3.6, line 6). Each *ObjectG* indicates one part of the robot (the RCX or wheel). Figure 3.8 shows each *ObjectG* that is relevant to the code in Figure 3.6. Each *ObjectG* has two properties, *Body* for dynamics simulation and *Geom* for collision detection. Note that when a *Body* is created using the function *dBodyCreate(World)*, it does not have any physics description until the functions *dBodySetPosition* (line 11) and *dBodySetMass* (line 13) define them. After defining the dynamic *Body*, the *ObjectG* can move around in the virtual world, but still does not have a bounding surface for collision detection. This is created by the geometry function *dCreateBox* (line 14) for the RCX and *dCreateCCylinder* (line 19) for the wheels. Both of the physical and geometrical functions must use the same dimensions as they both refer to the same *ObjectG*, and so the collision engine can obtain the correct position and orientation of the object from the *Body* as the simulation runs.

After the definition of each part of the robot, a hinge joint is used to attach each wheel to the RCX. The function, *dJointCreateHinge* (line 22) creates a hinge joint (*Joint*),
and $dJointAttach(Joint, Body1, Body2)$ builds a hinge joint ($Joint$) between the RCX ($Body1$) and the wheel ($Body2$) (see line 23). The anchor (see Figure 3.8) is the connection point of the two bodies. It is defined using the function, $dJointSetHingeAnchor(Joint, x, y, z)$. It is located at position ($x, y, z$), which is returned by the function $dBodyGetPosition$. The hinge joint allows a wheel to rotate around one specified axis (the axle of the wheel) defined by the function $dJointSetHingeAxis(Joint, 0, 0, 1)$ where $(0, 0, 1)$ indicates the Z-axis in world coordinates.

Figure 3.8 The robot with three wheels model

In ODE terms, each type of joint has a built-in motor, and the developer can decide whether to use it for the simulation. For example, a hinge joint for the front wheel in Figure 3.8 has a built-in motor located at its anchor. The function $dJointSetHingeParam (Joint, dParamVel, speed)$ can apply the motor with a type of velocity ($dParamVel$) on a specified hinge joint ($Joint$). The argument $speed$ indicates the value of the velocity. $dParamVel$ defines either an angular or linear velocity, which depends on the type of joint. In the case of the hinge joint shown in Figure 3.8, it is an angular velocity with units of radians per second. $LURobotSim$ can assign or change the value of the $speed$ based on the ROBOLAB program using the robot controller as shown in Figure 3.2. Note that if the motor is not to be used, the developer should not call the function $dJointSetHingeParam$ with the above arguments.
3.5.3 Environment

$LURobotSim$ has three pre-defined environments (E1, E2, and E3) as mentioned in section 3.5.

- Environment E1

The flat plane is treated as part of the static environment and is created in collision space. Thus it can collide with other objects but its position can not change with time. To create a flat plane, $LURobotSim$ uses the function, $dCreatePlane(Space, 0, 1, 0, 0)$ (see line 3 in Figure 3.6) in which $Space$ is the defined collision space. The next four parameters are the coefficients $(a, b, c, d)$ of the equation of the plane as shown in (3.1).

$$ax + by + cz = d$$

(3.1)

Thus, in $LURobotSim$, the equation for E1 is $y = 0$.

- Environment E2

The second pre-defined environment adds a four-wall surround to the flat plane (see Appendix F – Figure F.12). Each wall is used to test for collision simulation for a touch sensor. In this case, each wall is specified as a box defined in world coordinates. This allows the position of each wall to be set for rendering and collision detection. The position of a box is described using the function $dBodySetPosition(Body, x, y, z)$ that specifies the centre of the box at position $(x, y, z)$ in world coordinates. For example in Figure 3.9, one wall is defined at the position $(0, 0.1, 1.6)$.

When each wall is attached to a static environment (non-moveable object), it also becomes part of the static environment. To achieve this, a fixed joint, defined by the function $dJointCreateFixed$, is used to maintain the fixed position and orientation between a wall and the (non-moveable) plane as shown in Figure 3.9. The function $dJointAttach(Joint, Body1, 0)$ attaches each wall ($Body1$) to the plane (0). The argument, zero, means a non-movable object.
- **Environment E3**

The third environment adds a surrounding black line to the white flat plane for testing a light sensor. The black line is not implemented via ODE, but OpenGL. Section 3.6.3.2 discusses this in more detail.

```c
//the number of objects and joints is continued to the code
//in Figure 3.6

//create one side of the wall
dBodySetPosition(ObjectG[4].Body, 0, 0.1, 1.6);
dMassSetBox(&m, DENSITY, 3.2, 0.2, 0.03);
ObjectG[4].Geom[0] = dCreateBox(Space, 3.2, 0.2, 0.03);
dBodySetMass(ObjectG[0].Body, &m);
dGeomSetBody(ObjectG[4].Geom[0], ObjectG[4].Body);

//create the other sides
... //similar to above except the position

//connect four sides of the wall to the ground
for (int i = 4; i < 9; i++){
    Joint[i] = dJointCreateFixed(World, jointgroup);
dJointAttach(Joint[i], ObjectG[i].Body, 0);
dJointSetFixed(Joint[i]); }
```

**Figure 3.9** Example of defining four sides of the wall

### 3.6 Robot Controller

After setting up the TCP/IP connection and choosing a pre-defined robot, *LURobotSim* is able to receive instructions from ROBOLAB to generate the actions of the robot. This section describes the robot controller that *LURobotSim* uses to

- **store** the LASM instructions in a collection
- **read** the instructions in sequence
- **process** the instructions by sending information to ODE

The robot controller may be thought of as a virtual machine.
3.6.1 Storage of LASM instructions

In order to execute the LASM instructions, *LURobotSim* needs an appropriate data structure in which to store them. The controller uses an array list collection so that it can not only access each instruction in sequence but also access a particular instruction in the collection.

Figure 3.10 shows the structure of the controller. It consists of the array list as introduced above, and a sorted list to store address information. This is discussed in more detail in 3.6.2.

![Figure 3.10](image)

**Figure 3.10** The virtual machine structure

When the controller stores an LASM instruction, it stores the instruction as an object that has two string properties, *name* and *parameters*. As each LASM instruction is executed, the controller passes the necessary information to ODE functions to generate the actions of the robot. For instance, the LASM instruction, “dir 2, 2”, has “dir” as *name* and “2, 2” as *parameters* stored in the array list. When the controller executes this instruction, it passes the two parameters into a procedure that uses ODE functions to generate the correct actions, namely “Motor B running forward”. To carry out all the actions corresponding to an LASM program, *LURobotSim* uses a loop to read and process one LASM instruction at a time.

After connecting to ROBOLAB, the controller stores all LASM instructions in the array list in sequence. Table 3.5 shows the LASM instructions array list based on the program in Table 3.3 in section 3.3.
The index of the array list is the location at which the controller stores an LASM instruction. In addition, a sorted list (see Table 3.6) is constructed which stores each label name as key and its corresponding index in the array list as its value. The role played by the sorted list is explained in 3.6.2.

**Table 3.5** The array list stores all LASM program instructions

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Label0:</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>set</td>
<td>0,47,2,5</td>
</tr>
<tr>
<td>2</td>
<td>mulv</td>
<td>47,2,20</td>
</tr>
<tr>
<td>3</td>
<td>set</td>
<td>5,0,0,47</td>
</tr>
<tr>
<td>4</td>
<td>out</td>
<td>2,1</td>
</tr>
<tr>
<td>5</td>
<td>wait</td>
<td>2,200</td>
</tr>
<tr>
<td>6</td>
<td>Label1002:</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>chkl</td>
<td>2,50,1,9,0,Label1003</td>
</tr>
<tr>
<td>8</td>
<td>jmpl</td>
<td>Label1002</td>
</tr>
<tr>
<td>9</td>
<td>Label1003:</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>out</td>
<td>1,1</td>
</tr>
<tr>
<td>11</td>
<td>jmpl</td>
<td>Label0</td>
</tr>
</tbody>
</table>

**Table 3.6** The sorted list

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label0</td>
<td>0</td>
</tr>
<tr>
<td>Label1002</td>
<td>6</td>
</tr>
<tr>
<td>Label1003</td>
<td>9</td>
</tr>
</tbody>
</table>
3.6.2 Interpretation of LASM instructions

After storing the LASM instructions in the array list and the sorted list, *LURobotSim* uses a loop (see section 3.1.1) to call the controller in order to access each instruction one at a time. It then calls the relevant ODE functions in order to generate the actions of the robot. For example, when the controller reads the instruction named “jmpl” at index 8 in Table 3.5, it directs the loop to “Label1002”. The location of this label is stored in the sorted list as the value associated with the key “Label1002” (see Table 3.6) so that the loop transfers control to the instruction at index 6. This is done using a global variable named `indexcount` to store the current location in the array list. Usually, this variable is increased by one after reading each array list element. The controller reads and processes one LASM instruction during one iteration of the simulation loop (see Figure 3.1).

The “wait” instruction is used to suspend task execution for a given period. For example in Table 3.5 the “wait” instruction will suspend execution for two seconds. This is indicated by the last parameter, “200”, with units of 10 milliseconds. The controller does not increase the value of the `indexcount` in this case, so that it does not read and process the next LASM instruction until two seconds have passed. As a result, the robot will continue with the same action until two seconds have elapsed.

3.6.2.1 Motor Power Calibration

In ROBOLAB, users can set different power levels for a motor so that the motor rotates at different speeds. There are five different power levels as shown in Table 3.7. The default power setting is the full power level 5. As mentioned in section 3.5.2, to simulate the actions of a robot ODE needs to assign a value to the angular velocity of the motor on each driving wheel.

Referring to Table 3.1 in section 3.3 there are four LASM instructions (from index 4 to 7) to turn on a motor. These are described as follows.
set
The first “set” instruction at index 4 is used to assign the power level value (5) to the variable (47). The second one at index 6 is used to send the value of the variable (47) to the output port A (indicated by the first 0 in the instruction).

mulv
The “mulv” instruction at index 5 is used to multiply the value of the variable (47) by a constant value (20), since the motors expect a power value within the range [0, 100] rather than [0, 5]. The result of the calculation indicates not only the power value for turning on a motor but also the direction (a negative value indicates backward whereas positive indicates forward).

out
The “out” instruction at index 7 turns on (2) the motor at port A (1). It can also be used to turn off (1) a motor at a specified port (1, 2 or 3).

<table>
<thead>
<tr>
<th>Power Level</th>
<th>Angular velocity (radians/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.4</td>
</tr>
<tr>
<td>2</td>
<td>20.4</td>
</tr>
<tr>
<td>3</td>
<td>22.3</td>
</tr>
<tr>
<td>4</td>
<td>24.3</td>
</tr>
<tr>
<td>5</td>
<td>25.1</td>
</tr>
</tbody>
</table>

Thus, LURobotSim needs to convert the power levels in ROBOLAB to the angular velocity needed by ODE. Table 3.7 shows the angular velocity corresponding to each power level as measured using a test motor. This motor was not under load, and was directly connected to an angle sensor. In this case, the angle sensor measures the ________________

6 The default value for full motor power is displayed within a ROBOLAB program as 7. However, a user input value must be within the range [0, 5] and the distance that a robot travels with the default power value (7) is the same as it travels with the power value (5). As a result, LURobotSim uses the power value (5) as the full power value.
number of rotations of the motor every two seconds. From this the angular velocity can be calculated. It has units of radians per second. Appendix E provides detailed instructions on how the angular velocity was measured for different power levels.

With the power calibration, the controller is able to determine the angular velocity for a particular power level retrieved from the LASM instruction “set”, and the direction of the velocity by the LASM instruction “mulv”. The controller then calls the ODE function `dJointSetHingeParam (Joint, dParamVel, speed)` (see section 3.5.2) to turn the motor either on or off according to the LASM instruction “out”. In the off case a brake action is applied to the motor.

Note that in the case of a two-motor robot model, such as M3 (see Figure F.8 in Appendix F), motors drive the two front wheels. The controller turns on one motor before the other as it processes only one LASM instruction during each iteration of the loop. However, users will not recognize this in terms of the display of the robot movement, because the time step between each iteration of the loop is 1 millisecond. See section 3.9 for details of loop timing.

### 3.6.3 Sensors

Sensors allow a robot to respond to its environment. For example, a touch sensor allows a robot to detect a wall, and turn back (possibly) when the sensor hits it. Thus, the touch sensor is used to detect a collision between the robot and other objects in the environment. A light sensor is used to detect a light variation. This allows a robot to complete more complex tasks such as the line follower (see Figure 1.7). This section discusses the implementation of these two types of sensors, and the interpretation of LASM instructions that activate them.
3.6.3.1 Touch sensor

As mentioned in section 3.5, LURobotSim has a pre-defined robot with a touch sensor attached at the front of the robot (see Figure F.7 in Appendix F). In addition, LURobotSim has a pre-defined environment that has four walls that define an enclosed area (see Figure F.12 in Appendix F).

The ODE library has a built-in collision detection system that is described as a “space” (see section 3.5.1). In each simulation step, LURobotSim uses the ODE function, dSpaceCollide to determine which pairs of geometries in the space may potentially collide. It passes all geometries in the collision space to a call-back function, nearCallback (see Figure 3.11) to test for a potential collision between each pair of geometries via the ODE function dCollide. If there is a collision, this function will calculate the number of contact points. The ODE function dJointCreateContact then creates a contact joint based on each contact point. Referring to section 3.5.1, a contact joint applies an outgoing velocity in the direction of the contact normal (see Figure 3.7) for each pair of geometries to prevent them from intersecting. Each contact body has a contact point and an outgoing velocity is applied to each body in opposite directions.

Figure 3.11 shows example code for the collision detection system. LURobotSim uses it to detect a collision between the touch sensor (ObjectG[6]) and the wall (ObjectG[7]). When the touch sensor collides with the wall, the dCollide function in nearCallback will generate the number of contact points with outgoing velocity applied so that the touch sensor will not go through the wall. The status of the “push” VI will be changed to off.

In ROBOLAB, users use the sensor VIs, such as “Wait for Push” or “Wait for LetGo”, to control a touch sensor. These VIs use the LASM instruction “chkI” to compare the returned sensor value with a constant value, either zero or one. LURobotSim uses a global variable to hold the sensor value, for example, “pushed” for “Wait for Push” VI, and “released” for “Wait for LetGO” VI. The initial value of each variable is zero.
In the case of “Wait for Push”, the controller keeps checking the sensor value until the touch sensor detects another object in the space (the robot collides with an object). The ODE collision call-back function changes the value of the global variable to one, and the condition in the “chkl” instruction evaluates to false so that the controller jumps to the next LASM instruction.

```c
void nearCallback(void *data, dGeomID geom1, dGeomID geom2) {
    dBodyID body1 = dGeomGetBody(geom1); // get the body with geometry 1
    dBodyID body2 = dGeomGetBody(geom2); // get the body with geometry 2
    // define up to MAX contact points (in this case MAX=4)
    dContact contact[4];
    // set up friction parameter mu based on each contact point
    // see [17] for more friction parameters and descriptions
    for (int i = 0; i < 4; i++) {
        contact[i].surface.mode = dContactMu2;
        contact[i].surface.mu = dInfinity; // mu for F-dir1 (see Figure 3.7)
        contact[i].surface.mu2 = dInfinity; // mu for F-dir2
    }
    // collision test returns the number of actual contact points
    // or zero if there are none
    if (int numc=dCollide(geom1,geom2,4,&contact[0].geom,sizeof(dContact))){
        // if the ROBOLAB program uses a touch sensor and the pre-
        // defined robot model has a touch sensor (model 2)
        if (displaytouchsensor && model == 2) {
            // if the body 1 is sensor (ObjectG[6]) and body 2 is a
            // wall (ObjectG[7])
            if (body1 == ObjectG[6].Body && body2 == ObjectG[7].Body){
                pushed = 1; // change the sensor value to 1 to turn off highlight
            }
        }
    }
    // to add each contact point found to our contact group
    for (i = 0; i < numc; i++){
        // create a contact joint
        dJointID c = dJointCreateContact(World,contactgroup,contact+i);
        // attach to these two contact bodies with the joint
        dJointAttach(c, body1, body2);
    }
}
```

**Figure 3.11** Example code for collision detection
Figure 3.12 Example of collision detection program

The ROBOLAB program shown in Figure 3.12 allows the robot to detect a collision as it is going forward. If the touch sensor detects a collision, the robot will stop for one second and go backward for two seconds and then stop. Some of the LASM instructions are listed in Table 3.8. The “chkl” instruction at index 6 will check if the constant value (1) is greater than the sensor value. Initially, the sensor value is 0 so the controller will continue to the LASM instruction at index 7 which will lead the controller back to index 5. When the sensor detects a collision, the sensor value should be 1 so the condition evaluates to false, and the controller jumps to the address according to the label (Label1003) at index 8 and stops motor A at index 9.

Table 3.8 Parts of the LASM instructions for collision detection

<table>
<thead>
<tr>
<th>Index</th>
<th>LASM instructions</th>
<th>VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>set 0,47,2,5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mulv 47,2,20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>set 5,0,0,47</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>out 2,1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Label1002:</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>chkl 2,1,0,9,0,Label1003</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>jmp Label1002</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Label1003:</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>out 1,1</td>
<td></td>
</tr>
</tbody>
</table>

To implement the “Wait for LetGO” VI, the controller uses the same approach but with the opposite condition check in the “chkl” instruction compared with “Wait for push” VI.
3.6.3.2 Light sensor

*LURobotSim* implements a light sensor that is attached to a three-wheel robot (see Figure F.10 in Appendix F). The light sensor allows the pre-defined robot to perform tasks such as the line follower (see Figure 1.7). This section describes the approach that *LURobotSim* uses to implement the light sensor.

The environment for testing the light sensor is shown in Figure F.13 in Appendix F. It is generated using OpenGL texture mapping. A bitmap\(^7\) [50] image (e.g. the black line on a white background) is stored as a texture. The texture is stored in an array called the image data array. *LURobotSim* then maps it onto a square area on the flat plane.

Note that in ROBOLAB, the light sensor VIs such as “wait for dark” and “wait for light” check the input values from the sensor based on light variation which is reflected from the ground. *LURobotSim* simulates this by checking pixel greyscale values.

The simulation of the light sensor requires *LURobotSim* to repeatedly obtain the texel\(^8\) [51] value from a texture map. To achieve this, the light sensor faces the ground. *LURobotSim* obtains the position of the light sensor in world coordinates via the ODE function `dGeomGetPosition`. *LURobotSim* uses the position to compute the corresponding location in the image data array that is used to store all the texel values.

The example shown in Figure 3.13 shows the mapping between world coordinates and texture coordinates. The mapping area in world coordinates is shown as the top left square area, and the texture is shown as the top right square area. When the mapping area is created in world coordinates using the four vertexes (the four corners), each corner of the texture is assigned to each vertex so the texture is mapped

---

\(^7\) A bitmap is a type of image file format used to store digital images. The bitmap file stores all image pixels as byte format. Each pixel consists of three byte values representing red (R), green (G), and blue (B). Grey scale value has \(R = G = B\) in the range \([0, 255]\).

\(^8\) A texel is the texture element of texture space, used in computer graphics. Textures are represented by arrays of texels, as pictures are represented by arrays of pixels.
onto the area (the bottom textured area). For instance, the corner \((0, 0)\) in texture coordinates is assigned to the vertex \((x_a, y_a)\) in world coordinates.

![Diagram of mapping between world and texture coordinates](image)

**Figure 3.13** Example of mapping between world and texture coordinates

To retrieve a texel value from the image data array, *LURobotSim* uses the following approach. The position of the light sensor \((x_0, y_0)\) is used to calculate the corresponding location \((dx, dy)\) in texture coordinates via the equations 3.2 and 3.3 below.

\[
dx = (x_0 - x_a) \times \frac{1}{x_b - x_a}\]  

\[
dy = (y_0 - y_a) \times \frac{1}{y_b - y_a}\]  

where:

\(x_a\) is the left boundary of the mapping area in world coordinates

\(y_a\) is the bottom boundary of the mapping area in world coordinates
For instance, if \( x_a = -1, y_a = -1, x_b = 1, y_b = 1 \), and \((x_0, y_0) = (0, 0)\) is the origin in world coordinates, the corresponding location \((dx, dy)\) in texture coordinates is calculated as \((0.5, 0.5)\) which is the centre of the texture (see Figure 3.13). After calculating the position in texture coordinates, \(LURobotSim\) is able to retrieve the texel value from the image data array via \(dx\) and \(dy\). i.e. the array index is calculated by the integer part of \(N*dx\) and \(M*dy\).

After retrieving the texel value from the texture map, \(LURobotSim\) passes it to the controller to compare with the threshold value (see Figure 3.14) that is defined by the LASM command “chkl” (see for example Table 3.3). In ROBOLAB, the sensor VIs, “wait for dark” VI and “wait for light” VI are used to detect whether the light value is darker or lighter then a particular preset value. Both of them use the “chkl” instruction in a similar way to that of the touch sensor VIs as described in section 3.6.3.1.

![Figure 3.14 Example of line follower program](image)

Figure 3.14 shows the line follower program for testing the light sensor model, which is similar to the program shown in Figure 1.8.

In the case of the “wait for dark” VI, the controller uses the “chkl” instruction (see index 2 in Table 3.9) to check if the constant value (threshold value) is less than the returned sensor value (the retrieved pixel value from image data array). When the result evaluates to false, the controller will jump to the address according to the label “Label1003” (see Table 3.9), otherwise it will keep reading the sensor value.

In the case of “wait for light” VI the “chkl” instruction at index 11 checks if the threshold value is greater than the sensor value.
Table 3.9 Parts of the LASM instructions for line follower

<table>
<thead>
<tr>
<th>Index</th>
<th>LASM instructions</th>
<th>VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Label1002:</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>chkl 2,40,1,9,0,Label1003</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>jmp1 Label1002</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Label1003:</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>out 1,1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>set 0,47,2,5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>mulv 47,2,20</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>set 5,0,0,47</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>out 2,4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Label1004:</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>chkl 2,55,0,9,0,Label1005</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>jmp1 Label1004</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Label1005:</td>
<td></td>
</tr>
</tbody>
</table>

Initially, the pre-defined robot moves on the white area (see Figure F.13 in Appendix F), so the light sensor returns the pixel value 255. The controller passes this value into the “chkl” instruction for “wait for dark” VI. The threshold value (40) is less then 255 so the controller will not process the next instruction for “stop A” VI. Thus the robot will keep moving until it reaches the black line. At this time, the threshold value (40) is greater than the returned sensor value (0) so the condition evaluates to false, and the controller will process the “stop A”, “turn on motor C” and “wait for light” VIs.

The “wait for light” VI has the opposite condition check than “wait for dark”. The robot will keep turning with motor C until the light sensor reaches the white area. Then it will keep turning with motor A looking for the black line. Repeatedly turning on motor A and motor C will keep the robot following the black line.

Note that the threshold values in the program are representative of the values that may be needed by an actual robot to follow the line. The values required for an actual robot may vary depending on actual environment and lighting conditions.
Alternatively, the “Wait for Darker” VI and the “Wait for Lighter” VI shown in Table 3.10 can be used to perform the line follower task. In this case, the threshold value used by the “chkl” instruction is a variable rather than a constant value (in the case of “Wait for Dark” and “Wait for Light”). For example, in the case of “Wait for Darker”, initially, the sensor (9) value on port 1(0) is assigned to the variable (47) (see index 1 in Table 3.10). The threshold value is then calculated by the “subv” (subtract) instruction. It subtracts a constant value (40) from the variable (47) at index 2. The source type is indicated by the value (2). The comparison via the “chkl” instruction is the same as that of the “Wait for Dark” VI.

<table>
<thead>
<tr>
<th>Index</th>
<th>LASM instructions</th>
<th>VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>set 0,47,9,0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>subv 47,2,40</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Label1002</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>chkl 0,47,1,9,0,Label1003</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>jmpl Label1002</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Label1003</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>set 0,47,9,0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>sumv 47,2,55</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Label1004</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>chkl 0,47,0,9,0,Label1005</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>jmpl Label1004</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Label1005</td>
<td></td>
</tr>
</tbody>
</table>

The case of “Wait for Lighter” is similar to “Wait for Darker” except that it uses “sumv” instruction to calculate the threshold value, which adds a constant value to the variable (47). Note that the “Wait for Darker” and “Wait for Lighter” VIs have not been implemented in the current version of LURobotSim.
3.7 OpenGL Rendering

Referring to section 3.5.1, ODE builds the physics and geometrical descriptions for each component of the robot and its environment, but ODE does not make it visible to the user. As outlined in section 3.2.4, OpenGL functions render the robot and its actions based on the physical and geometrical descriptions from ODE. This section discusses the rendering in more detail.

3.7.1 The rotation matrix conversion from ODE to OpenGL

In OpenGL each vertex of every object to be rendered is multiplied by the 4x4 modelview matrix [25]. The matrix determines how the vertices of an object are transformed from world (object) coordinates to viewing (camera) coordinates. Geometric transformations such as rotation, translation and scaling are applied to objects in a scene via this matrix. The last column of the modelview matrix contains the current position vector of the object relative to the camera’s position. The top left 3x3 sub-matrix contains the scaling and rotation operations.

Before using OpenGL to render the scene, the modelview matrix must be constructed based upon the ODE rotation matrix. LURobotSim defines and uses the function ODEtoOGL to form the modelview matrix by storing the elements of the ODE rotation matrix in column major order and adding in the position vector (p[0], p[1], p[2], 1), retrieved from ODE using dGeomGetPosition (see Figure 3.15 and Figure 3.16).

\[
\begin{bmatrix}
M_0 & M_4 & M_8 & M_{12} \\
M_1 & M_5 & M_9 & M_{13} \\
M_2 & M_6 & M_{10} & M_{14} \\
M_3 & M_7 & M_{11} & M_{15}
\end{bmatrix}
\quad
\begin{bmatrix}
R_0 & R_1 & R_2 \\
R_3 & R_4 & R_5 \\
R_6 & R_7 & R_8 \\
0 & 0 & 0
\end{bmatrix}
\]

a) OpenGL modelview matrix 

b) ODE rotation matrix

Figure 3.15 Matrices in ODE and OpenGL
Figure 3.16 The conversion from the ODE rotation matrix to the OpenGL matrix

Figure 3.16 shows the function where M is an array that stores the OpenGL modelview matrix, p is the position vector returned by the ODE function `dGeomGetPosition` and R is the ODE rotation matrix returned by the ODE function, `dGeomGetRotation`.

### 3.7.2 Render functions

After the creation of the modelview matrix, OpenGL draws each component of the robot based on the geometrical descriptions from ODE, using the function `DrawGeom` (see Figure 3.17).

```c
void ODEtoGGL(float* M, const float* p, const float* R) {
}
```

Figure 3.17 Example of a rendering function using ODE data

`DrawGeom` calls ODE functions to obtain geometry information such as the dimensions, position vector and rotation matrix of the object, and then renders the
object calling the relevant OpenGL function, such as \textit{DrawBox(sides, pos, R)}. These arguments represent the geometry information; “sides” defines the dimensions, “pos” defines the position vector in world coordinates and “R” defines the ODE rotation matrix. Before calling OpenGL functions to render the objects, the \textit{DrawBox} function calls the \textit{ODEtoOGL} function as described in section 3.7.1.

3.8 Tracking program execution

As described in section 3.2.5, another feature of \textit{LURobotSim} is tracking program execution. Users can see which VI is currently executing. This code will correspond to a particular action of the virtual robot. The idea is similar to the ROBOLAB code-highlighting feature (see Figure 3.18). In ROBOLAB, icons can be highlighted as they execute. When the “bulb” button is turned on, the icons are highlighted during and after execution of the associated VI while the rest of the icons remain greyed out. This feature is helpful for debugging a program provided it is not to be downloaded to the RCX. For example, it may be an image processing ROBOLAB program for displaying an image captured from a live camera.

\textbf{Figure 3.18} Example of ROBOLAB highlighting feature

Unfortunately, this feature does not help users follow program execution when programmed for the RCX, because the code–highlighting feature is executed when the
program is downloaded into the RCX, not when the program runs within the RCX. For instance, when the “waiting for 2 seconds” icon is highlighted using the above code-highlighting feature, the program does not wait for two seconds, it is simply being downloaded to the RCX for later execution, at which time it causes a 2-second wait.

However, *LURobotSim* does provide this facility by highlighting each icon as the corresponding VI executes during the simulation. Each icon from ROBOLAB is stored and then displayed by *LURobotSim* as a texture image. Before the simulation starts, all the ROBOLAB icons are read from graphic files and stored in an array called TEXTURE. *LURobotSim* then reads the LASM instructions into its array list and sorted list as described in section 3.6.1. After this, the texture ID (i.e. index in TEXTURE) for each icon is identified and stored in an array called ORDER by reading the stored LASM instructions in sequence.

When the simulation starts *LURobotSim* displays each icon in ORDER using OpenGL texture mapping, and subsequently highlights each icon as the corresponding VI is executed by the controller.

In summary, to track and display program execution as the simulation runs, there are three steps as described below.

- Read all LASM instructions and store each corresponding icon in the ORDER array as an object that has a texture ID as its property. The texture ID refers to the corresponding texture image in the TEXTURE array.

- Render all icons used in a ROBOLAB program by reading each icon stored in the ORDER array using OpenGL texture mapping. Note that this step needs to be repeated during each iteration of the loop. This is because *LURobotSim* renders the scene using double buffering [25], and each time the buffers are swapped, the icons need to be redisplayed.

- Highlight the icon as the simulation runs according to the VI being executed as determined by the LASM instruction currently being executed. The highlighting
feature is done by rendering an icon for a second time. In this case, the mapping uses a brighter colour compared with the other displayed icons.

Figure 3.19 presents the pseudo-code for tracking program execution.

```
- Initialise the TEXTURE array that defines all ROBOLAB icons
- Obtain and store the texture ID for all program icons in the ORDER array
- While (program not finished)
  - Display all icons stored in the ORDER array using OpenGL texture mapping
  - Call controller to process the stored LASM instructions to obtain the index in the ORDER array of the VI currently being executed
  - Use OpenGL to display current executed VI based on the index in the TEXTURE array
```

**Figure 3.19** The program structure for rendering program execution trace

To view the rendered icons, *LURobotSim* provides two options for users: billboarding that displays the icons in the same display area as the robot, and Multiple Document Interface (MDI) that displays the icons in a separate window. The following sections discuss these two options.

### 3.8.1 Billboarding

Billboarding is a technique that makes a flat object (billboard) face a target, where the target may be the camera, another object, or a 3D location. In this project, the target is the camera.

The execution trace is therefore generated as a 2D object but displayed in a 3D representation area. Initially, it faces the camera so that users can view it. When users change the viewpoint, they should also be able to view the 2D trace properly. Hence *LURobotSim* adjusts the 2D display orientation so that it always faces the camera. i.e. so its normal vector is always in the opposite direction to the camera viewing direction (see Figure 3.20).
The approach requires the OpenGL modelview matrix to be of the form shown in Figure 3.21, i.e. the top left 3x3 sub-matrix is set to the identity matrix to align the coordinate system of the camera to the world coordinate system. This means the billboard display will not have any rotation or scaling when it is rendered.

\[
\begin{bmatrix}
1 & 0 & 0 & M_{12} \\
0 & 1 & 0 & M_{13} \\
0 & 0 & 1 & M_{14} \\
M_3 & M_7 & M_{11} & M_{15}
\end{bmatrix}
\]

**Figure 3.21** The modified modelview matrix for billboarding

To implement billboarding, *LURobotSim* follows the following steps.

- Save a copy of the current OpenGL modelview matrix to the matrix stack using the OpenGL function, *glPushMatrix*.
- Reset the top 3x3 sub matrix to the identity matrix.
- Load the matrix to the OpenGL state machine using the function, *glLoadMatrixf*.
- Render the object in the world coordinates using the user defined function *DrawIcon*.
- Restore the original modelview matrix using the OpenGL function, *glPopMatrix*. 
Figure 3.22 shows the results at different viewpoints generated by billboarding.

![Figure 3.22 Example of billboarding displays](image)

3.8.2 Multiple Document Interface

The other approach for displaying program execution is to use a separate window tiled with the 3D display window that shows the actions of a robot. To achieve this, *LURobotSim* uses a multiple document interface (MDI). In MDI terms, the main window has a menu toolbar and is called the frame window. This is the only window in a single document interface (SDI) program. The frame window may have additional windows in an MDI program. In this project there is one MDI client window and a toolbar window. These two windows are child windows of the frame window. Child windows are placed in the client area of the frame window (see Figure 3.23). In addition, each child window has its own client area where a program may draw images and place controls.

The MDI client window differs from the toolbar window in that it can have child windows whereas the toolbar window cannot. In this project, two additional windows
are defined as children of the MDI client window. One displays program execution and the other displays the actions of the robot. These are displayed in the client area of the MDI client window. Figure 3.24 shows the general structure of this MDI window application.

![Diagram of MDI window application](image)

**Figure 3.23** Example of MDI windows and its components

![Diagram of window structure](image)

**Figure 3.24** The general structure of an MDI window application

A user controlled event, such as clicking a button or selecting a menu item, has a pre-defined identity, stored as a windows message. The procedure corresponding to this event is executed in the call-back function for the frame window. Note that the main windows call-back function is defined only for the frame window. The two MDI child windows (two display windows) have their own windows call-back functions to
process windows messages. In this case, the frame window call-back function does not process the windows messages from these two child windows, but only from their parent window (the MDI client window). *LURobotSim* uses the Windows API to program the MDI application. For a detailed programming tutorial on creating a windows application using the Windows API, the reader can refer to the FunctionX Win32 programming tutorial in [48]. Note that in this study the OpenGL Utility Toolkit (GLUT) was considered as an alternative to the Windows API. GLUT is a window system independent toolkit that may be used to display the output from OpenGL. However, some incompatibility issues with ODE were found.

Figure 3.25 shows a screenshot of the MDI version of the simulator. It shows the toolbar window and two MDI child windows. The upper MDI child window displays the program execution trace in 2D while the lower one displays the actions of a robot in 3D.

![Figure 3.25 Example of Multiple Document Interface display](image)

In order to allow users to interact with a simulation, *LURobotSim* uses a GUI including menus (in frame window), buttons (in toolbar window) and display windows (see Figure 3.25). The toolbar buttons allow the user to carry out tasks such
as starting a simulation, changing a viewpoint, and adjusting the simulation speed. For a detailed explanation of the user interface, the reader should refer to the user guide provided in Appendix F.

### 3.9 Simulation Time control

By default *LURobotSim* will generate and display a simulation of the actions of a robot at normal speed, i.e. as near as possible to real time. In addition, as briefly mentioned in section 3.2.5, the *pause* and *slow down* controls may be used when the actions are too fast to view. Users are able to view more details by slowing the simulation speed. This section discusses how the simulation time is related to real time, and how the simulation time controls work. Section 3.9.1 explains the simulation timing in normal mode (normal speed), which is followed by the explanations of slow mode and pause mode.

*LURobotSim* defines several variables relevant to the timing of the simulator. They are described as follows.

- **simulation time**

  The simulation time step as used by *LURobotSim* is the time difference between consecutive simulation states. *Simulation time* stores the total simulation time since the start of the simulation.

- **current time**

  *Current time* holds the real time obtained from the computer’s operating system.

- **target time**

  *Target time* stores the time that the next frame should be displayed so that the simulation output is displayed to the user in real time. It is based on the number of simulation steps, N, between frames.
Figure 3.26 shows the general timing structure. More detailed explanations are given in the following sections. Note that this structure applies only to normal mode. For slow and pause modes, see sections 3.9.2 and 3.9.3.

![Figure 3.26 The timing structure](image)

### 3.9.1 Normal mode

*LURobotSim* uses a constant simulation step size of 0.001 seconds in order to make the simulation accurate and stable. In practice, the smaller the simulation step the more reliable and stable the simulation. This value was obtained by comparing a simulation result with the result from a real situation. The comparison is based on the distance that the robot travels in two seconds. Initially, the value was set as 0.01 second, and the simulation result was not accurate, because the distance that the virtual robot travelled was much further than the real robot travelled. By decreasing the value, the accuracy of the simulation was increased. When the value is 0.001 the simulation result was similar to the real one with about one percent error.

However, with a simulation step of 0.001 second if *LURobotSim* displays every simulation step, the simulation is displayed in slow motion. In this case, there are too many calculations from both ODE and OpenGL in each iteration of the loop so that one simulation time step calculation and display can not be completed in real time. To address this problem, *LURobotSim* displays every fiftieth ODE simulation step. This figure was determined by experimenting with different numbers of simulation steps.
between each display using a computer that has Celeron® 2.4G processor, Intel® “Extreme Graphics 2” graphic card and 1 G memory. It balances the requirements of running a simulation in real time and displaying the actions of the robot smoothly. Using the same computer *LURobotSim* could display the actions of the robot in real time, but not smoothly, when the number of steps between displays was increased to 100.

The *current time* is updated after the ODE calculations in order to make real time synchronise with simulation time if necessary. See Figure 3.26. When the *current time* catches up with the *target time*, *LURobotSim* displays the next frame. Otherwise, it waits until the *current time* becomes greater than or equal to the *target time*. Thus, for example, a faster computer processor takes less time to process a simulation step than a slower one so that it is more likely to have to wait until the *current time* matches the *target time*. With a slower processor *LURobotSim* may not wait for the synchronisation because the *current time* may always be greater than the *target time* after the simulation step calculations during each iteration of the loop. In this case, the simulation will always run slower than real time. Hence, a faster processor may be able to display the actions of a robot in real time whereas the slower processor may always display them slower than real time.

### 3.9.2 Slow mode

If the user wishes to slow down a simulation, *LURobotSim* applies a delay in calculating successive simulation steps. It uses the Windows function called *Sleep(time)*. It is shown in Figure 3.27 between simulation steps. Note that the ODE simulation time step does not change, so the actions of a robot are the same as the actions in normal mode. In this case however, if *LURobotSim* were to display every fiftieth step, the actions of a robot would not appear smooth. This is because *LURobotSim* has increased the actual time between each step. Therefore, when in
slow motion, \textit{LURobotSim} displays a frame after every ODE simulation step instead of every fiftieth step.

![Timing structure for slow mode](image)

**Figure 3.27** The timing structure for slow mode

### 3.9.3 Pause mode

To make a simulation pause, \textit{LURobotSim} applies a Boolean variable “\textit{checkStop}” to control the ODE simulation step. Its default value is false. When it is set to true, \textit{LURobotSim} repeatedly calls the OpenGL functions to draw the current state of the robot without the simulation time advancing. This repeated rendering still allows users to interact with the display if necessary. Users are able to change the value of the “\textit{checkStop}” through the GUI so that they can cause the simulation to pause or resume normal processing.

### 3.9.4 Implementation Summary

Referring to the general simulation loop structure in Figure 3.1, Figure 3.28 shows the detailed pseudo-code that includes the simulation timer issues discussed above.

Note that the \textit{current time} is updated in each iteration of the outer loop using the Windows timer function, \textit{QueryPerformanceCounter()}. In slow mode, \textit{LURobotSim} uses the Windows function \textit{Sleep (time)} to delay the simulation step. The delay time can be controlled by the user via the GUI. The delay time is increased by 0.01 second each time the user clicks the button for slow mode. \textit{LURobotSim} has 0.2 second as the maximum delay time. In addition, the pause mode can be also controlled via GUI. The button for pause mode is used to change the value of “\textit{checkStop}”. 

73
3.10 Programming environment

Although the OpenGL and ODE libraries are developed using the C++ language, and have a C++ interface, they can also be used with other language environments such as VB and C# using the Tao Framework [40]. The Tao Framework for .NET allows developers to use the OpenGL API or ODE API in a VB or C# application by using a DLL (Dynamic Link Library)⁹ that includes OpenGL or ODE.

In this project, LURobotSim is programmed using a C++ programming environment. The C/C++ header files of the two libraries are included at the beginning of the

---

⁹ A DLL (Dynamic Link Library) is a small application (or sometimes a group of them) that is called on by a larger application to provide a service or set of instructions.
program. This project uses OpenGL version 1.1. ODE version 0.7 is used which is available for downloading at [17].

Microsoft Visual C++ .NET 2003 is the particular programming environment used. This comes with an OpenGL library. Appendix B presents a detailed installation and configuration guide for using ODE in the Microsoft Visual C++ .NET 2003 environment.

3.11 Summary

This chapter presents the techniques that have been used to implement LURobotSim, and discusses relevant issues that arose during the development. It presents not only an overall structure of LURobotSim, such as the system overview and architecture in section 3.1 and section 3.2, but also details such as the physics description using ODE in section 3.5, the robot controller in section 3.6, rendering the scene using OpenGL in section 3.7, and tracking and display of the ROBOLAB program execution in section 3.8.

At the start of the implementation of LURobotSim the basic behaviour of a pre-defined LEGO robot, such as turning one motor forward for a certain time was used. Afterwards sensors were implemented. As explained in section 3.6.3, LURobotSim is able to display more complex actions of the pre-defined robots, such as a line follower using a light sensor and detection of other objects using a touch sensor.
Chapter 4 Discussion

This chapter discusses successes and limitations of *LURobotSim* by analyzing the results from several tests and the feedback from several users. These tests are discussed in section 4.1 and mainly test the accuracy and functionality of *LURobotSim* on basic tasks such as going forward and turning as well as more complex tasks such as line following. In addition, educators who use LEGO MINDSTORMS to teach robotics tested *LURobotSim* in terms of usefulness and usability. Their feedback is summarised in section 4.2, which includes the positive features and some limitations of *LURobotSim*. Some other limitations are then discussed in section 4.3.

4.1 Tests of accuracy and functionality

Several tests were developed using different pre-defined robot models as described in Appendix F (see Figure F.6 to Figure F.10). Note that all these tests assume the environment is ideal so that, for example, the wheels and ground have sufficient friction so that the robot will not slip.

- Basic functionalities

![Figure 4.1](image)

*Figure 4.1* The grids for measuring the distance the robot travels

With regard to basic functionalities such as turning on motors to move forwards, or backwards, with different power levels, the distance that the virtual robot travels is
measured using the grid area as shown in Figure 4.1. The grid spacing represents ten centimetres in the physical world.

Table 4.1 shows the comparison of simulation results and real world measurements. This indicates the distances that the robot travels with different motor power levels. The data are measured with the ROBOLAB program “go forward for 2 seconds and stop” using the robot model M4 (see Figure F.9 in Appendix F). The real world robot is built to be the same as the virtual model, with four wheels attached to the RCX, and the two front wheels as the driving wheels.

Distances are measured in the real world situation using a ruler. In the simulation, the distances are calculated by subtracting the RCX’s initial position value (0) from the stopping position value. The results for the virtual robot are very similar to the real world situation. Note that in the real world situation, the motor power is provided by six AA batteries. As a result, if the batteries have a reduced power level, the results may vary from the data shown in Table 4.1.

Table 4.1 Comparison of the distance travelled in 2 seconds

<table>
<thead>
<tr>
<th>Power Level</th>
<th>Distance(metre) ((LURobotSim))</th>
<th>Distance(metre) (\text{real world robot})</th>
<th>Relative Difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.82</td>
<td>0.81</td>
<td>1.2</td>
</tr>
<tr>
<td>2</td>
<td>0.86</td>
<td>0.84</td>
<td>2.3</td>
</tr>
<tr>
<td>3</td>
<td>0.94</td>
<td>0.93</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>1.02</td>
<td>1.01</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>1.06</td>
<td>1.05</td>
<td>0.9</td>
</tr>
</tbody>
</table>

The robot model M3, with a back wheel that is free to turn, (see Figure F.8 in Appendix F) is used to test the turning function. The ROBOLAB program for turning is shown in Figure 4.2.

![Figure 4.2 ROBOLAB program for testing turning function](image-url)
The two motors drive in opposite directions for two seconds to turn the robot. Table 4.2 shows the comparison of the angle turned by the RCX within the simulation and the real world. In the real world situation, the data are the average of five measured angles for each power level. The number of complete revolutions of the RCX is counted visually, and the extra angle turned is measured using a protractor. In the simulation, the angle is obtained from the rotation angle in the modelview matrix [25] ($M_0$ in Figure 3.15a in section 3.7.1).

According to the data shown in Table 4.2, the results of the simulation with power levels 1, 4, and 5 are similar to those of the real world robot. The simulations with power levels 2 and 3 have larger differences compared with the real world robot. The reason for this is not clear. Note that in general, because a real world environment may provide extra friction to the turning wheels, or because the construction of the real world robot may not be exactly the same as the virtual one, different results may be obtained.

Note that the results may also vary depending on the condition of the RCX batteries.

**Table 4.2 Comparison of the angle turned in 2 seconds**

<table>
<thead>
<tr>
<th>Power Level</th>
<th>angle (revolution) ($\text{LURobotSim}$)</th>
<th>Average angle (revolution) (real world)</th>
<th>Min angle (revolution) (real world)</th>
<th>Max angle (revolution) (real world)</th>
<th>Relative difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.289</td>
<td>2.249</td>
<td>2.236</td>
<td>2.250</td>
<td>1.7</td>
</tr>
<tr>
<td>2</td>
<td>2.408</td>
<td>2.277</td>
<td>2.250</td>
<td>2.305</td>
<td>5.4</td>
</tr>
<tr>
<td>3</td>
<td>2.631</td>
<td>2.311</td>
<td>2.292</td>
<td>2.319</td>
<td>12.2</td>
</tr>
<tr>
<td>4</td>
<td>2.860</td>
<td>2.808</td>
<td>2.681</td>
<td>2.847</td>
<td>1.8</td>
</tr>
<tr>
<td>5</td>
<td>2.950</td>
<td>2.949</td>
<td>2.903</td>
<td>2.983</td>
<td>0.03</td>
</tr>
</tbody>
</table>

- **Touch sensor**

For testing the touch sensor, the pre-defined robot model M2 (see Figure F.7 in Appendix F) is used to collide with a wall (E2, see Figure F.12 in Appendix F). The ROBOLAB program as shown in Figure 3.12 is used to program the robot actions.
The result shows that the touch sensor is able to detect the collision between the sensor and the wall, and the robot executes the programmed reaction (i.e. stop for one second and go backwards for two seconds and then stop).

- **Light sensor**

For testing the light sensor, the virtual robot model M5 (see Figure F.10 in Appendix F) is used to detect greyscale colour variation as mentioned in section 3.6.3.2. The ROBOLAB program is shown in Figure 3.14. It uses a loop to repeatedly check greyscale colour variation as the robot moves along a black line (E3, see Figure F.13 in Appendix F). Note that the colour is set by OpenGL.

The sensor successfully detects brightness variation when the robot drives from the white area to the black area. Then the robot turns off motor A and turns on motor C so that it turns and keeps driving forward until the sensor detects the white area. At that time, the robot turns on motor A and turns off motor C so that it drives back to the line. The robot repeats the above processes to keep following the line.

### 4.2 Feedback from Educators

Several educators who use LEGO MINDSTORMS for teaching robotics were asked to test *LURobotSim* and provide feedback. The educators include the Director of the CEEO\(^{10}\) at Tufts University, education staff at Science Alive!, Christchurch\(^{11}\), and a teacher at Hagley College, Christchurch\(^{12}\).

---

\(^{10}\) Centre for Engineering Educational Outreach (CEEO) is the organization that develops the ROBOLAB software to combine with LEGO MINDSTORMS to teach students programming and engineering concept (see detail at http://www.cceo.tufts.edu/, retrieved at 28.11.07).

\(^{11}\) Science Alive!, Christchurch uses LEGO MINDSTORMS toolkits to teach school students fundamental robotics concepts as well as learning to build and program a robot in a robotic class (see detail at http://www.sciencealive.co.nz/robotics/robotics.htm, retrieved at 28.11.07).

\(^{12}\) Hagley College, Christchurch teaches student programming skills in programming logic using a range of robotics toolkits (see detail at http://computing.hagley.school.nz/about, retrieved at 28.11.07).
4.2.1 Positive features

The overall concept of *LURobotSim* is seen to be useful by these educators as either an extra resource to assist teaching with LEGO MINDSTORMS in an educational environment or as a useful substitute if users do not have access to LEGO hardware. Positive features that were identified include the following:

- Highlighting program execution is particularly good for students to understand the programming logic when the program executes. This becomes even more useful when students learn more complex programming logic, such as the line follower.

- *LURobotSim* is seen as useful for giving demonstrations to entire classes, and showing students what their program should do before they start to program, without actually showing them the ROBOLAB program code, e.g. how the line follower should work. This can be done by showing the actions of a robot with the code highlighter switched off.

- In a robotics lab, there may be not enough physical environments for students to use simultaneously to test their programs, e.g. for a robot that is programmed to stop on a black line, Science Alive! has about 12 PC workstations, but only 3 “black line” models. Students could test their program using *LURobotSim* while they wait for a model to become available.

- Many children, particularly younger ones, are keen to start building a real world robot as soon as possible, without thinking about program logic. *LURobotSim* is a useful tool to let the children concentrate first on the programming logic. The building aspect can then follow once the logic is clearly understood.

- With larger groups of students, *LURobotSim* offers students the opportunity to work in teams, with some students building the robots while others write the program and check the code using *LURobotSim*. 
4.2.2 Limitations

There are some drawbacks of LURobotSim that were pointed out by these educators.

- **Realism of Rendering**

Because LURobotSim uses only simple primitive shapes such as boxes and cylinders to represent the pre-defined robots, the display of the robot may be lacking in realism. More details of the robot, such as motors, axles and other LEGO pieces have not been included. This may cause users to be less interested in the simulation, although the educators thought this was not a major limitation. To them the behaviour of the robot was more important.

- **User interface**

The user interface needs to be further developed. For example, there is no tool tip for each button. This would make it easier to know the function of each of them. In addition, users may expect more functions that allow them to interact with LURobotSim more easily. For example, users may want to change the environment by drag and drop operations. For example, an environment could be changed interactively by adding a barrier that a robot that is already in motion, has to negotiate.

- **The robot controller cannot execute all LASM instructions**

Currently, LURobotSim is at an early stage of development and does not implement all ROBOLAB VIs.

Some VIs may be implemented relatively easily, such as the “Wait for Darker” and “Wait for Lighter” VIs used for light sensor detection (see section 3.6.3.2). These are discussed as future developments in section 5.2.4.

Other VIs use LASM instructions that are similar to some that have been implemented. For the controller this does not cause a problem, since it simply executes the corresponding LASM instructions. However, it may be difficult to
distinguish between these different VIs when displaying the corresponding icons. For instance, consider the “Touch sensor fork” VI and the “Wait for push” VI (see section 3.6.3.1). Figure 4.3 shows an example of the use of “Touch sensor fork” VI. It checks if the returned touch sensor value is less than 1 using the “chkI” instruction, and turns on motor A if the result evaluates to true, or turns on motor B if false. The “Wait for push” VI has only one output wire. It does not offer a choice.

![Diagram](image)

**Figure 4.3** Example program using the “Touch sensor fork” VI

Currently the “chkI” instruction is used to identify the correct icon to be displayed. However, both VIs use similar parameter values for this instruction (see Table 4.3). Checking subsequence LASM command may solve this problem.

<table>
<thead>
<tr>
<th>VI</th>
<th>LASM instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Touch sensor fork”</td>
<td>chkl 2,1,0,9,0,Label</td>
</tr>
<tr>
<td>“Wait for push”</td>
<td>chkl 2,0,1,9,0,Label</td>
</tr>
<tr>
<td></td>
<td>chkl 2,1,0,9,0,Label</td>
</tr>
</tbody>
</table>

A further limitation involves multiple tasks. In *LURobotSim*, the controller can process only one task at a time. As a result, for more complex programs such as monitoring two sensors at the same time using the “Task split” VI, as shown in Figure 4.4, the controller processes only the second task (checking for the touch sensor on port 2), and ignores the first task (checking for the light sensor on port 1). As a result, the virtual robot can test for only one sensor reacting with the environment at a time.

---

13 In ROBOLAB, a task may be represented by the sequenced LASM instructions. The overall structure of a task begins with the LASM command “task” or “start”, and is terminated by the command “endt”. Figure 4.4 shows two tasks in one program, and each task includes several VIs.
In addition, the pre-defined robot cannot interact with another pre-defined robot because each simulation can have only one robot.

![Diagram of ROBOLAB program for task split]

**Figure 4.4** Example of ROBOLAB program for task split

- **Multiple views of simulation**

*LURobotSim* does not provide multiple views, from different viewpoints, simultaneously. The use of multiple windows to display these different viewpoints would be a useful feature to assist with understanding the actions of a robot.

### 4.3 Other Limitations

*LURobotSim* is a useful tool to give users an understanding of what will happen in certain conditions, and allow them to test their ROBOLAB program before downloading it into an RCX. However, simulations can never be exactly the same as the real world situation. For instance, light variation and the variation in friction on different surfaces may influence the behaviour of the real world robot.

For the version of *LURobotSim* reported in this thesis, the simulation is limited to the simple actions of a vehicle with one or two motors. Although *LURobotSim* can accurately simulate most basic cases, there are several other limitations described as follows.

- **Robot models and environments**

Currently *LURobotSim* is not designed for users to build their own virtual robots and environments. This can only be done by giving users access to the source code, and
then they must have the knowledge to change the code. Hence, for most users there are just the pre-defined robots and environments. LURobotSim can test the ROBOLAB program only with these pre-defined models in the given environments.

- **Light sensor**

LURobotSim limits the light sensor to greyscale values so it can detect only greyscale texture images. Although the virtual robot successfully follows a line (black on white background), the real world robot is different from the virtual one in that it detects light variation, reflected from the ground rather than the colour variation.

- **Program tracing**

The program tracing feature does not show parameter values or interconnections between icons. For example, Figure 4.5a) shows a ROBOLAB program with parameter values shown. This is compared with the program tracing display shown in Figure 4.5b) with the “Wait for 2 seconds” icon highlighted.

![a) Example ROBOLAB program](image)

**a) Example ROBOLAB program**

![b) Example of program tracing display of LURobotSim](image)

**b) Example of program tracing display of LURobotSim**

**Figure 4.5** Comparison of a ROBOLAB program with the program tracing display

- **Other issues**

Although LURobotSim displays waiting time for “Wait for seconds” VI using OpenGL texture mapping (see Figure 4.6), it does not display other quantitative measures, such as displaying the distance that a robot travels from start point to end
point. This may be a useful feature to give more detailed information about the simulation to the user.

**Figure 4.6** Example of the display of wait time
Chapter 5  Summary and Future work

5.1 Summary

The goal of this project was to create a simulator for LEGO MINDSTORMS robotics, to be used with ROBOLAB software. It simulates the actions of pre-defined LEGO robots so that users can test their ROBOLAB programs before downloading them into LEGO hardware. In addition, it allows users who do not have the hardware to learn ROBOLAB programming skills which may then be tested and debugged using the simulator. Tracking program execution also helps the user to learn and understand programming logic. We accomplished this goal by using ODE as the physics engine to simulate the actions of a robot, and OpenGL as the rendering tool to make the simulation visible to users. Once a program is debugged, any difference in the actions of the real world robot compared with the simulation must be caused by either robot construction or the real world environment.

The ODE physics simulation is separate from the rendering procedure. In each iteration of the simulation loop, ODE calculates one time step of the robot’s action. Rendering functions use the ODE physics simulation to provide the position, orientation and dimension of each object in order to render correctly.

As discussed in chapter 4, although LURobotSim is able to simulate and display the actions of a robot accurately in an “ideal” environment, and was considered by those who evaluated it, to be a useful tool, it currently has some limitations. Future improvements are discussed in section 5.2.
5.2 Future developments

In order to improve *LURobotSim* according to the limitations discussed in chapter 4, this section discusses future enhancements related to the GUI, model construction, 3D representation and virtual robot functionality.

5.2.1 GUI enhancement

In order to make the graphical user interface more friendly and easier to use, this section suggests some enhancements.

- **Tool tips**

  For each button on the toolbar, a tool tip should be generated to inform users of the functions of this button. This can be done by updating the current Windows application to a .NET application, because it is easier to apply a tool tip to a button in the .NET environment. The tool tip is a property of each button object.

- **Program tracing**

  Currently, *LURobotSim* does not display the parameters for some VIs, such as the power level for turning on a motor or the input port used for a sensor. This information is stored in LASM instructions. The power level parameter is provided by the “set” instruction, and the input port parameter is provided by the “sent” instruction. Note that Appendix A provides the explanations of these instructions in more detail. After they are defined, OpenGL could render these parameters as images on screen using texture mapping.

  Currently, the interconnection between icons is not displayed. To improve this, *LURobotSim* could draw a thick line, using OpenGL, between the icons that are displayed as shown in Figure 5.1.

![Figure 5.1 Example of interconnection of icons](image_url)
In ROBOLAB each VI has input terminal(s) and output terminal(s) for connection to other VIs. The connection wire should connect one output terminal of a VI to one input terminal of the other VI. Figure 5.2 shows an example of input and output terminal(s) of VIs. The first VI at left has one input terminal (top left square) and two output terminals (right hand side), the second VI has one input (top left) and one output terminal (top right), and the third VI has two input terminals (left hand side) and one output terminal (top right). The green square in the first VI connects a Boolean variable. The blue square in the second VI connects a parameter. The white square indicates no connection.

**Figure 5.2** The input and output terminal(s) of VIs

When each icon is stored by the controller as an object, the terminals of the VI could also be stored as properties of an icon object. The properties of a terminal may include its ID and the 2D location (x, y) in world coordinates (the centre location of a terminal, see Figure 5.2). Note that the position of the connection line between two icons would be identified by the location of the terminals of the two icons.

In the case of each of the “Motor” and “Wait For” VIs, there is one input and one output terminal so that each icon can be connected using a horizontal thick line (see for example in Figure 5.1). In the case of multiple tasks (see Figure 4.4), and forks (see Figure 4.3), the terminals used by each icon would be identified before rendering the connection line. If the locations of the terminals that are connected have the different y values, the connection wire would be a series of connected straight lines, e.g. “L shape”. 
Multiple simulation views

In order to allow users to view a simulation from different viewpoints simultaneously, multiple windows could be used, one for each viewpoint. The variable “g_iMaxChild” is used to limit the number of child windows that LURobotSim can have. Currently it is set at two. To implement multiple window views, LURobotSim could increase this value, and create a camera for each of these child windows.

5.2.2 3D representation

The 3D display of a robot may not look realistic compared to the real world robot, because LURobotSim uses only simple primitive shapes to represent each part of a pre-defined robot, and the virtual robot may lack some detail, such as motors and axles. Although 3D representation was not seen by the people who evaluated LURobotSim as an important issue, compared with the robot functionality (see detail in section 4.2.2), simple modifications to improve this could be made relatively easily.

Extra pieces, which are not based on the physics description from ODE, may be applied to each part of the robot, because the rendering functions are independent of the ODE simulation. For example, the RCX is described as a box in ODE. OpenGL can add more shapes to the ODE box to make the RCX look more realistic. Note that these shapes are not defined in ODE as parts of the RCX, so that they do not have physical representation. This leads to consideration of how a robot, or environment, can be modified, or a new one constructed.

5.2.3 Visual construction of the robot and the environment

Currently in LURobotSim, the InitODE procedure (see Appendix D) is used to define all properties of a pre-defined robot and its environment. The DrawModel procedure (see Appendix D) is then used to render each part of the robot and its environment. The physical properties of a robot such as mass, position, and dimensions need to be specified in the InitODE code, and the rendering commands for each component are
specified in the *DrawModel* code. This process is not done interactively. An interactive approach would involve visual construction.

Visual construction of a robot may be an extremely complex task. It is suggested that further developments of *LURobotSim* should start with some relative easy tasks such as interactively adding an object into the scene. A possible approach for this feature is discussed below.

For a robot or environment that has only simple shapes, such as the pre-defined models described in this thesis, visual construction may be done using software such as BlockCAD\(^{14}\) shown in Figure 5.3 and LEGO Digital Designer\(^{15}\). It allows users to visually construct a LEGO object with many LEGO pieces. Everything in the software can be controlled with the mouse. Users can click on a LEGO piece image in the collection on the right hand side, and click at a point in the display area where the user wants to place the piece.

![BlockCAD user interface](http://web.telia.com/~u16122508/proglego.htm)

**Figure 5.3** BlockCAD user interface

\(^{14}\) BlockCAD is a freeware program for building virtual models with Lego-like bricks. http://web.telia.com/~u16122508/proglego.htm, retrieved at 24.10.2007

\(^{15}\) LEGO Digital Designer is a free software and is available at http://ldd.lego.com/download
To improve *LURobotSim* based on this feature, *LURobotSim* could create a library that defines all components that may be used to build a robot or an environment. It also defines the properties of each component include initial position, orientation, dimension, and mass. In the case of construction of a robot, the components should be LEGO pieces. In the case of the environment, the components may be LEGO pieces or other objects. Each component may have an image (icon) as a component of the GUI so that users can select it. *LURobotSim* could define a new robot class. When users want to create their own robot, they could click a button that may be called “user defined model” so that *LURobotSim* generates a new robot object that would be defined in the *InitODE* procedure, and constructed from one or more LEGO pieces (see Figure 5.4).

![Diagram](image)

**Figure 5.4** User-defined robot and its components

After a user selects a LEGO piece and defines its position in the 3D display area using a mouse, *LURobotSim* could create a new part object for the robot and store its mass, position, dimensions and orientation according to the information provided in the component library. When two selected objects are in contact, the joint between them (a property of the robot) may be defined by supplying the necessary information, including the type of the joint and the joint location, using a form in the GUI. This information would be properties of the joint. The scene tree editor (see Figure 2.5) used in Webots in section 2.1.2.1 could be a useful additional tool to assist users to define each part of a robot.
The updated system architecture is shown in Figure 5.5 where users can select a pre-defined model or define their own.

![Figure 5.5 Updated system architecture](image)

However, the above simple approach is limited. For detailed and more complex robot construction, other considerations need to be taken into account. These include:

- The port on RCX, to which a motor or a sensor is connected, needs to be defined interactively.
- The friction between a robot and its environment needs to be specified when a new robot and environment are created.
- The 3D location of a LEGO piece needs to be defined.
- The joined pieces that must make contact (e.g. engine gears) or must not make contact (turning wheel should not touch RCX) need to be specified.

These issues would not be easy to implement using the above software. However, the reader can refer to software such as Deep Creator™[^16] that allows the user to visually construct a complex 3D object in detail. This software may provide some hints for further enhancements for the visual construction issues.

5.2.4 Enhanced Robot functionality

To further improve LURobotSim, the functionalities of the pre-defined robots need to be enhanced so that the robot is able to perform more complex tasks. This would make LURobotSim a more useful resource to debug more complex programs.

- Implementation of more light sensor VIs

In LURobotSim the controller processes all LASM instructions. The LASM instructions of “Wait for Darker” and “Wait for Lighter” VIs are similar to the “Wait for Dark” and “Wait for Light” except that they have two extra instructions, “set” and “subv” (for the “Darker” case) or “mulv” (for the “Lighter” case). These two instructions are used to set a variable value to be used as a threshold to compare with a returned sensor value. The “subv” instruction subtracts a given value from a variable, and stores the result in that variable. The variable is used as a threshold to compare with the sensor value at run time. To implement this in LURobotSim, the “subv” instruction needs to be added into the controller (see section 3.6) to perform a subtraction operation. To implement the “Darker” or “Lighter” VIs, the “chkI” instruction in the controller needs to be updated by adding the variable as the threshold value instead of a constant value as used by the “Dark” or “Light” VIs (see section 3.6.3.2).

- Improvement of light sensor functionality

Simulating light variation is a complex task in OpenGL. To implement light variation, one possible approach may be to use the HSV\(^{17}\) colour model, standing for Hue, Saturation, and Value (brightness). The HSV coordinate system is cylindrical, and the model is defined as a cone (see Figure 5.6).

\(^{17}\) HSV describes colours as points in a cylinder whose central axis ranges from black at the bottom to white at the top with neutral colors between them, where angle around the axis corresponds to “hue”, distance from the axis corresponds to “saturation”, and distance along the axis corresponds to “value” (brightness) [52].
By converting the RGB value to the HSV value, \textit{LURobotSim} could retrieve the brightness value based on the V value. This indicates the brightness of a colour. V is in the range 0 to 1. For example, V = 0 represents black, and for saturation = 0, V = 1 represents white and V = 0.5 represents grey.

Table 5.1 shows different (V values) for the same hue (green). Note that the hue value is 120 degrees, and the saturation value is 1 for all cases. For the conversion formula from RGB to HSV, the reader can refer to [52].

\textbf{Table 5.1} The green colour representation with different lightness value

<table>
<thead>
<tr>
<th>Colour</th>
<th>V = 1</th>
<th>V = 0.75</th>
<th>V = 0.5</th>
<th>V = 0.25</th>
<th>V = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td><img src="image" alt="Green V=1" /></td>
<td><img src="image" alt="Green V=0.75" /></td>
<td><img src="image" alt="Green V=0.5" /></td>
<td><img src="image" alt="Green V=0.25" /></td>
<td><img src="image" alt="Green V=0" /></td>
</tr>
</tbody>
</table>

Hence when \textit{LURobotSim} retrieves an RGB value from a texel, it could convert this value to HSV so that its brightness value can be compared with the threshold value (see section 3.6.3.2). Note that before the brightness value is used, it must be scaled to the range 1 to 100, which is the range used by the light value in the light sensor VIs.

- Multiple tasks

To implement simultaneous multiple tasks, such as monitoring two sensors at the same time, as shown in Figure 4.4, \textit{LURobotSim} could define a task class, and each task would be defined as an object of this class. A task would have its LASM
instructions as a property. All tasks would be stored in a collection (see Figure 5.7). When the simulation starts, the controller could read and process one LASM instruction from each task, and update the simulation time after it has processed all tasks. Note however that each task may also have a priority attribute so that low priority tasks must wait until high priority tasks have been executed. This would also need to be taken into account. This approach still processes one task at a time, but the sequential execution does not matter as simulation time would not be updated between tasks.

Note that the executed VI in each task would be highlighted at the same time. This is because the icon for the executed VI would be rendered after the simulation time is updated.

![Figure 5.7 The multiple tasks structure](image)

**5.2.5 Summary**

This section discusses possible enhancements to *LURobotSim*. They are related to the limitations in the four main areas addressed in chapter 4. For the GUI, the display of program execution may also display connection between icons. For the display of a robot, extra pieces may be rendered to be more realistic. Virtual construction would allow users to build their robot more flexibly and easily. Enhancement of robot functionality would be achieved after implementing more Vis, and multiple tasks, in order to allow the virtual robots to perform more complex actions.
References


[10]. LEGO Mindstorms Scout, User Guide & Reference


http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=383760&isnumber=8697


[15]. Quick Start Guide to computer control and robotics using LEGO MINDSTORMSTM for Schools (2004) The LEGO Group retrieved from online
http://www.lego.com/eng/education/mindstorms/home.asp?pagename=qsg&l2id=4_4&id=4_4_2&id=4_4_2_1

[16]. Rogers, C.B.(2008), Personal Communication


[27]. http://www.sdsc.edu/~nadeau/Courses/Siggraph98vrml/vrml97/vrml97.htm, retrieved at 12.05.2007

[28]. http://orbit.medphys.ucl.ac.uk/, retrieved at 20.05.2007


[34]. http://www.robocup.org/, retrieved at 20.05.2007
[37]. http://www.continuousphysics.com/Bullet/, retrieved at 20.05.2007
[38]. http://ox.slug.louisville.edu/~o0lozi01/opal_wiki/index.php/Main_Page, retrieved at 20.05.2007
[40]. http://www.taoframework.com/about, retrieved at 23.05.07
[42]. http://www.tzi.de/4legged/bin/view/Website/WebHome retrieved at 25.05.07
[45]. http://www.w3.org/TR/REC-xml/, retrieved at 2.06.07
[46]. http://alex.ais.fraunhofer.de/zeno/web?action=content&rootid=17536 retrieved at 2.06.07
[47]. http://en.wikipedia.org/wiki/Friction, retrieved at 10.08.07
[48]. http://www.functionx.com/win32, retrieved at 01.02.07
[50]. http://en.wikipedia.org/wiki/Bitmap, retrieved at 09.10.07
[51]. http://en.wikipedia.org/wiki/Texel_%28graphics%29, retrieved at 10.10.07
[53]. http://www.e-learningfoundation.com/, retrieved at 14.02.08
[54]. http://en.wikipedia.org/wiki/Rigid_body_dynamics, retrieved at 30.05.07
Appendix A

LASM instructions List

The following section lists all LASM instructions and parameter sources (see Table A.1) for this project in alphabetical order. Each instruction consists of the LASM syntax and its explanation. These instructions include program commands and direct commands. Note that some of the commands can be either program or direct commands according to [10]. However, this project uses them as only one type.

Table A.1 The sources used in LURobotSim

<table>
<thead>
<tr>
<th>Source</th>
<th>Type</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Variable</td>
<td>0-47</td>
<td>Variables are identified by unique numbers. Each variable stores a 16 bit signed value</td>
</tr>
<tr>
<td>2</td>
<td>Constant</td>
<td>-32768 - 32767</td>
<td>Different commands accept different actual values</td>
</tr>
<tr>
<td>8</td>
<td>Program Slot</td>
<td>0-4</td>
<td>Returned and accepted values are in the range 0 to 4.</td>
</tr>
<tr>
<td>9</td>
<td>Sensor value</td>
<td>0-2</td>
<td>The returned value is generated in the hardware, based on the sensor type and mode</td>
</tr>
</tbody>
</table>

LASM instructions:

chk1

Syntax:

chk1 src1, var1, relop, src2, val2, forward label

Explanations:

src1, src2 are source numbers as listed in Table A.1, var1 and var2 are the values corresponding to the sources, and relop is a condition (0: greater than, 1: less than, 2: equal to, and 3: not equal to). If the comparison fails (the condition evaluates to False), the execution jumps to the given address (Label), otherwise execution continues with the next instruction.

Type: program command
**decvjnl**

Syntax:

    decvjnl variable, label

Explanation:

    Decrement the variable and jump if the variable becomes negative. This is a way of implementing loops. The variable is the loop counter.

Type: program command

**dir**

Syntax:

    dir direction, motor list

Explanation:

    This command sets the motor status register with the given values,
    Direction: backward (0), forward (2); Motor list: port A(1), B(2), C(4).

Type: program command

**jmpl**

Syntax:

    jmpl label

Explanation:

    The command jumps to a certain Label, either forwards or backwards. Label represents a memory address in a program. An infinite loop can be implemented using the command “jmpl 0”.

Type: program command

**mulv**

Syntax:

    mulv variable, source, value

Explanation:

    Multiplies the given variable by the given value, and the result is stored in the variable.

Type: program command
**out**

Syntax:

```
out OnOff, motor list
```

Explanation:

The command sets the motor status register with the given values.

- **OnOff**: Off(1), On(2); Motor list: port A(1), B(2), C(4), or All(7).

Note that All(7) is used only for Off.

Type: program command

**plays**

Syntax:

```
plays sound
```

Explanation:

The command plays a given system sound, if sounds are globally enabled.

The sound is represented by a unique number in the range 0 to 5. For example, 1 indicates a beep sound.

Type: direct command (replay 0xA6)

**poll**

Syntax:

```
poll source, value
```

Explanation:

Reads the given source and value combination and sends the reply back.

When ROBOLAB communicates with the RCX, it sends this command to the RCX to check if the program slot is available using the source value (8) (see Table A.1), and expects a reply from the RCX.

Type: direct command (reply 0xE5)
prgm

Syntax:

  prgm program slot

Explanation:

  The command stops all tasks in the currently selected program and then
  selects the given program slot. There are 5 program slots in the RCX. If
  the command is executed as part of a downloaded program, task 0 is
  started in the new program.

Type: direct command (reply 0x66)

pwr

Syntax:

  pwr motor list, source, power value

Explanation:

  Sets the motor status registers with the new power levels. Source indicates
  the source type of the power value. See Table A.1.

  Motor list: port A(1), B(2), C(4).

Type: program command

sent

Syntax:

  sent sensor number, sensor type

Explanation:

  Sets the given sensor to the given type.

  Sensor number: 0-2 (the three input ports).

  Sensor type: 0 (None), 1(switch), 2(temperature), 3(reflection, used for
  light sensor), 4(angle).

Type: program command
**setv**

Syntax:

```
setv variable, source, value
```

Explanation:

Set the variable to have the given value. Source indicates the source type of the value. See Table A.1.

Type: program command

**set**

Syntax:

```
set destination source, destination value, origin source, origin value
```

Explanation:

The command is a general purpose command that can be used to set any writeable source inside the RCX to the contents of any other source.

Type: program command

**stop**

Syntax:

```
stop task number
```

Explanation:

The command stops the given task.

Type: direct command (reply 0x76)

**subv**

Syntax:

```
subv variable, source, value
```

Explanation:

The command subtracts the given value from the given variable, and the result is then stored in the variable.

Type: program command
wait

Syntax:

wait source, wait value

Explanation:

Suspends task execution for the given number of 10 ms. If the given number is negative, the command is ignored.

Type: program command
Appendix B

ODE configuration guide - Microsoft Visual Studio.NET 2003

Note that the information in this appendix is needed by developers who want to write an application using ODE with Microsoft Visual Studio.NET 2003. Some of the material provided in this guide can be found in the ODE document [17].

Developers need to place the library file “ode.lib” in the VC++.NET library folder with the path “..\Microsoft Visual Studio.NET 2003\vc7\platformSDK\lib” and the C++ header files in the include folder with path “..\Microsoft Visual Studio.NET 2003\vc7\platformSDK\include\ode”. When developers create a new VC++ project with MS Visual Studio.NET 2003, they need to build a link between the project and the ODE library. This is done as follows.

1) Open the project properties window under the project menu (see Figure B.1).

2) Open the “Input” property in the “Linker folder” (see Figure B.2).

3) Type “ode.lib” into the “Additional Dependencies” (see Figure B.3).

After configuration, developers can call ODE API functions through the header file “ode/ode.h”.

Figure B.1 Project properties menu
Figure B.2 Project link property

Figure B.3 Additional Dependencies
Appendix C

Winsock API library configuration guide - Microsoft Visual Studio.NET 2003

Note that some of the material provided in this section may be found in an Internet Winsock tutorial called “Winsock Programmer’s FAQ”\(^\text{18}\) and [41].

**Configuration**

Microsoft Visual Studio.NET 2003 has the built-in Winsock library, “ws2_32.lib”. Developers do not need to download it. However, if developers want to apply it in a new project, they need to include it in the link property. The configuration is the same as that of the ODE library described in Appendix B. Figure C.1 shows the library file included in the Additional Dependencies.

![Additional Dependencies](image)

**Figure C.1** Winsock library included in the Additional Dependencies

To implement a Winsock application, some programming details referred to in section 3.4.1 are presented below.

1. **Initialise the Winsock Library**

The initialise function, `WSAStartup ()`, checks if the Winsock library version is correctly installed on the computer before calling any other Winsock API functions. Otherwise, the Winsock API functions would not be recognized.

---

\(^{18}\) See detail at http://tangentsoft.net/wskfaq/, retrieved at 01.12.07
2. Create a TCP listening socket

After initialising the Winsock library API, the developer is able to create a TCP (Transmission Control Protocol) listening socket for communication using the function, `socket (af, type, protocol)`. The parameter `af` defines the address family, `type` defines the new socket type, and `protocol` defines the protocol to be used. In the case of protocol TCP, the parameter `af` should be `AF_INET`, `type` should be `SOCK_STREAM`, and `protocol` should be `IPPROTO_TCP`. All Winsock references are given in [41]. In addition, a socket structure called `SOCKADDR_IN` stores all server information. Figure C.2 presents the socket structure and its application.

```c
struct sockaddr_in  {
    short sin_family;  //Protocol type
    u_short sin_port;  //Port number of socket
    struct in_addr sin_addr;  //IP address
    char sin_zero[8];  //Unused
};

// Use a SOCKADDR_IN structure to fill in address information

SOCKADDR_IN serverInfo;

serverInfo.sin_family = AF_INET;

// Since this socket is listening for local connections, any local address will be ok
serverInfo.sin_addr.s_addr = INADDR_ANY;

// Convert integer 4266 to network-byte order with htons() and insert into the port.
serverInfo.sin_port = htons(4266);
```

**Figure C.2** Example of socket structure
3. **Bind the socket to the local server address**

After creating a new socket, the `bind (listening socket, server info, server size)` function is able to bind the new socket to the local server address that is defined in a socket structure. For example, `bind (LURobotSim, (LPSPACEADDR) &serverInfo, sizeof(struct sockaddr))` should bind the “LURobotSim” listening socket to the “serverInfo” server.

4. **Listen for the client**

After binding the new socket, the `listen (listening socket, backlog)` function is used to listen to the socket for a connection from the client.

For example, `listen (LURobotSim, 1)` allows the socket LURobotSim to listen for 1 client connection, which is indicated by the parameter 1.

5. **Accept a client connection request**

In order to accept a client’s connection request, the `accept (listening socket, server address, address size)` function needs to be used. The accept function returns a client socket. For example, `accept (LURobotSim, (struct sockaddr*) &from, &fromlen)`

After accepting a connection, the server should be able to communicate with the client using the `recv (listening socket, messages, length, 0)` function to receive messages and the `send (listening socket, messages, length, 0)` function to send messages.

As mentioned in section 3.3, when LURobotSim communicates with ROBOLAB, the direct commands expect a response from LURobotSim. There are a number of direct commands that are used during the communication process including “poll”, “prgm”, “stop”, and “plays”. For example, for the “poll” command, ROBOLAB expects a reply “e5 00 00” from LURobotSim, otherwise it will not transfer the rest of the LASM program. Appendix A lists specifications of the other direct commands relevant to this thesis.
6. **Finish and Close socket connection**

Once a client no longer requires a connection and has transferred all the LASM instructions, the server closes the listening socket for this client via the `closesocket (listening socket)` function. If the Winsock application is not to be used, the server can shut it down using the `WSACleanup()` function.

Note that the reader may refer to [41] for more Winsock application example codes.
Appendix D

Main procedures in the simulation loop

The simulation loop is in the main program function, WinMain. Before the simulation starts, the simulator calls the InitODE function using the main windows call-back function, MainWndProc (see Figure D.1). InitODE defines the physics objects that the user selects, including a pre-defined robot and its environment. This call-back function is used to handle the Windows messages for the windows program. The online Windows application tutorial in [48] provides more details about this.

![Diagram](image)

**Figure D.1** The main procedures in the simulation loop

The simulation and rendering functions are called independently. Controller and SimModel calculate the simulation step using the ODE function dWorldStep, and DrawModel to draw the current state of the selected pre-defined robot. To draw the object, the function DrawGeom calls ODE functions to obtain the physics information for the pre-defined object, and passes the information to the corresponding drawing function such as DrawBox, DrawCylinder that have OpenGL rendering instructions.
In addition, to display the ROBOLAB program execution, the function \textit{DrawTrace} renders the program icons, and the function \textit{DrawIcon} highlights a particular icon that represents the VI being executed. Both of these functions use OpenGL rendering instruction.
Appendix E

Power Calibration in ROBOLAB

In order to simulate the movement of a robot with a motor that has different power levels, we need to know the rotational speed of the motor corresponding to each power level. Figure E.1 presents a ROBOLAB program that was used to measure the angular velocity of a test motor. The program measures the angular velocity of the motor by using a container to hold the angle value measured by an angle sensor which connects to input port 1 of an RCX. The angle value is measured after the motor has been turned on for two seconds.

![Diagram](image)

**Figure E.1** Example of ROBOLAB program for testing angular velocity

The angle sensor in ROBOLAB returns the number of rotations (measured in sixteenths of a rotation). As a result, the number of revolutions of the angle sensor is given by (E.1), in which \( R \) is the number of revolutions, and \( S \) is the angle sensor value.

\[
R = \frac{S}{16} \quad (E.1)
\]

Then, the angular velocity (radians per second) is given by (E.2), in which \( V_a \) is the angular velocity, \( t \) is time, and \( R \) is the number of revolutions.

\[
V_a = \frac{2 \pi R}{t} \quad (E.2)
\]
Appendix F

User guide

1. Configuration for ROBOLAB connection

Firstly, users need to install ROBOLAB (version 2.9) with patch, 2.9.3.

Secondly, users need to create a text file as shown in Figure F.1 to assign the TCP/IP port (4266) to ROBOLAB, on which LURobotSim is listening. The IP address should be “localhost”, if ROBOLAB and LURobotSim run on the same computer.

Finally, users need to save this file as “sym.txt” in the ROBOLAB install folder.

2. Simulation Steps

After the configuration of ROBOLAB, users are able to transfer their ROBOLAB program to LURobotSim and run their simulation by clicking several buttons in the toolbar.

For all buttons and features, such as environment change, simulation time controls, and viewpoint control, refer to Table F.2 at the end of this appendix.

Alternatively, users can also use the menus to control the simulation.

For example, to simulate a single motor Lego robot, without a sensor, moving on a flat plane (default environment) for 2 seconds, users need to follow the steps as follows.
Program it using ROBOLAB (see Figure F.2).

Figure F.2 Example of ROBOLAB program

Select the correct COM port for communication with the simulator (See Figure F.3).

Figure F.3 The Select COM Port list

Run the simulator and click the button in order to start a new connection with ROBOLAB.

Click the button to open the window for displaying the actions of the robot.

Transfer the ROBOLAB program to the simulator by clicking the white arrow button (see Figure F.3). Users should be able to see Figure F.4 which indicates for a successful transfer.

Figure F.4 A successful connection between ROBOLAB and LURobotSim
In the simulator, select a pre-defined robot model using the button (see other buttons for the other robot models in Table F.2). To confirm or cancel the selection, users need to click the “OK” or “Cancel” button (see Figure F.5).

![Figure F.5 Confirm or Cancel a model selection](image)

Run the simulation using the button To view the program execution window, users need to use the button.

Click the button again when users want to load a new ROBOLAB program. Note that the new simulation still uses the previous robot model and its environment. However users can change them by clicking model buttons (see Table F.2).

The current version of LURobotSim can work with five pre-defined robot models and three environment models including as follow shown in Figure F.6 to F.11 below. Note that each back wheel of the model M1, M2, and M4 is connected to the RCX using an independent axle. However, the axles are on the same axis.
Single motor (port A) robots without touch sensor (button)

![Figure F.6 Robot model M1](image)

Single motor (port A) robots with touch sensor attached on port 1 (button)

![Figure F.7 Robot model M2](image)

Two motors (port A and port C) robot without touch sensor with 3 wheels

![Figure F.8 Robot model M3](image)

Two motors (port A and port C) robot without touch sensor with 4 wheels

![Figure F.9 Robot model M4](image)

Two motors (port A and C) robot with light sensor and 3 wheels

![Figure F.10 Robot model M5](image)
A Flat plane

Figure F.11 Environment model E1

A surrounding wall on a flat plane

Figure F.12 Environment model E2

A surrounding black line on a flat plane

Figure F.13 Environment model E3

The simulator is able to simulate the basic functionality of a LEGO robot, such as going forward or backward, turning around, touch sensor and light sensor. However, it is not able to simulate complex tasks such as monitoring two sensors at same time using task split.
3. **Mouse and keyboard controls**

Users can interact with the display using the keyboard and mouse. Table F.1 shows detailed specifications for each interactive control. The world coordinate is shown in Figure F.14.

![The world coordinates for 3D display](image)

**Figure F.14** The world coordinates for 3D display

**Table F.1** Specification for each interactive control using mouse and keyboard

<table>
<thead>
<tr>
<th>Device</th>
<th>Control Button</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse</td>
<td>left button</td>
<td>change camera angle anti-clock wise based on Z axis</td>
</tr>
<tr>
<td></td>
<td>right button</td>
<td>change camera angle clock wise based on Z axis</td>
</tr>
<tr>
<td></td>
<td>up arrow</td>
<td>move camera forward along positive X axis</td>
</tr>
<tr>
<td></td>
<td>down arrow</td>
<td>move camera backward along negative X axis</td>
</tr>
<tr>
<td></td>
<td>left arrow</td>
<td>move camera along positive Z axis</td>
</tr>
<tr>
<td></td>
<td>right arrow</td>
<td>move camera along negative Z axis</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>move camera down along negative Y axis</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>move camera up along positive Y axis</td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td>change camera angle clock wise based on Y axis</td>
</tr>
<tr>
<td></td>
<td>F4</td>
<td>change camera angle anti-clock wise based on Y axis</td>
</tr>
<tr>
<td></td>
<td>F5</td>
<td>move camera in program execution window along positive Z axis</td>
</tr>
<tr>
<td></td>
<td>F6</td>
<td>move camera in program execution window along negative Z axis</td>
</tr>
</tbody>
</table>
**Table F.2** Control buttons and explanations

<table>
<thead>
<tr>
<th>Buttons</th>
<th>Explanations</th>
</tr>
</thead>
<tbody>
<tr>
<td>🎁</td>
<td>Run the simulation</td>
</tr>
<tr>
<td>🎁</td>
<td>Pause the simulation</td>
</tr>
<tr>
<td>🎁</td>
<td>Speed up the simulation</td>
</tr>
<tr>
<td>🎁</td>
<td>Slow down the simulation</td>
</tr>
<tr>
<td>🎁</td>
<td>Stop the simulation</td>
</tr>
<tr>
<td>🎁</td>
<td>Tile display for multi-windows</td>
</tr>
<tr>
<td>🎁</td>
<td>Cascade display for multi-windows</td>
</tr>
<tr>
<td>🎁</td>
<td>Pre-defined robot model (one motor without sensor)</td>
</tr>
<tr>
<td>🎁</td>
<td>Pre-defined robot model (one motor with touch sensor)</td>
</tr>
<tr>
<td>🎁</td>
<td>Pre-defined robot model (two motor without sensor)</td>
</tr>
<tr>
<td>🎁</td>
<td>Pre-defined robot model (two motor without sensor)</td>
</tr>
<tr>
<td>🎁</td>
<td>Pre-defined robot model (two motor with light sensor)</td>
</tr>
<tr>
<td>🎁</td>
<td>Pre-defined environment model (surrounding wall)</td>
</tr>
<tr>
<td>🎁</td>
<td>Pre-defined environment model (line follower)</td>
</tr>
<tr>
<td>🎁</td>
<td>Open program execution display window</td>
</tr>
<tr>
<td>🎁</td>
<td>Open 3D display window</td>
</tr>
<tr>
<td>🎁</td>
<td>Replay the simulation</td>
</tr>
<tr>
<td>🎁</td>
<td>Build connection with ROBOLAB</td>
</tr>
<tr>
<td>🎁</td>
<td>Change viewpoint, F: robot’s front view, B: back, T: top, and S: right side</td>
</tr>
<tr>
<td>🎁</td>
<td>i: initial position</td>
</tr>
</tbody>
</table>