VIRTUAL REALITY SIMULATION FOR LEGO MINDSTORMS ROBOTICS

Sushmita Subramanian
Partner: Theo Brower
Fall 2003-Spring 2004
Advised by Chris Rogers and Michael Littman

Abstract

In this project, we use Virtual Reality Modeling Language (VRML) integrated with Robolab to create a prototype for software, which will simulate the movements of a Lego robot in the virtual world. Our project succeeded in meeting its goal of simulating a Lego robot moving forwards, backwards, and rotating for any amount of time, additionally simulating a robot using a light sensor or a push sensor to explore and react within an environment. We hope that the integration of virtual reality in this project will both challenge us and expand the possible applications of Lego Mindstorms by overcoming the physical constraints of the real world.

Introduction

Lego Mindstorms is a robotics toolkit that allows students to design and program Lego robots. The toolkit contains an RCX Microcomputer, Robolab software, motors, sensors, and many Lego components. Students create a robot by combining the RCX with motors, sensors, and other Lego parts, and then programming the robot with the included Robolab software. After creating a Robolab program, students load it into the RCX of their physical Lego robot using an infrared interface. They then run the program on the RCX (autonomous of the computer), and their robot moves based on their program.[1]

This programming process provides an excellent way for students to get hands-on experience learning basic concepts of engineering and robotics. Lego Mindstorms is currently being used to teach programming, math, science, engineering (the application of science and math to real world tasks), and much more. The kit allows teachers to integrate many different subjects into a fun and memorable project that students enjoy. While Lego Mindstorms has done an excellent job in teaching the above subjects, students should also be introduced to new-age tools which have been widely adopted by industry and academia. One such tool is the virtual reality simulation. By utilizing a simulation to design and program their Robolab programs, students will come to understand the value of being able to test their ideas on the computer without the real world

[1] For more information about Robolab and education see the Tufts Center for Engineering Outreach webpage at http://www.ceeo.tufts.edu/robolabatceeo/resources/articles/default.asp. A good introduction is "ROBOLAB: Intuitive Robotic Programming Software to Support Life Long Learning" by Merredith Portsmore

constraints of space and limited resources. Virtual reality allows students to quickly and easily test their understanding of difficult concepts; with the click of a button they can see if their program works. In addition, virtual reality programs enable students to "visit environments and interact with events that distance, time, or safety factors make unavailable."[2]

Virtual simulation has shown to be a very effective tool in both entertainment and education. Virtual reality has become an essential component of video and computer games, and educational software programs have been forced to utilize it in order to compete with video games for adolescents' attention. For example, there is now a virtual reality program called Superscape Virtual Reality Toolkit that is used to aid Computer Integrated Manufacturing education. It allows students to create interactive 3d worlds that can be published on the Internet. It also uses a range of editors to allow students to work on different parts of the virtual reality world constructions. The entertainment industry has been driving the development of virtual reality technology, but when it is integrated into educational activities, it provides an interesting way for students to take on difficult subjects. Recently, a number of programs have utilized virtual reality to introduce students to the subject of robotics. One such program allows elementary school students to learn basic artificial intelligence concepts by programming a soccer playing robot. The robot can be assigned to different tasks, such as playing soccer, avoiding an object, or wandering.[3] In each case, the student can see how the robot's behavior changes as a result of their commands.

Virtual reality is a very powerful tool that can be used to enhance a student's educational experience beyond what is normally possible. However, it is essential that virtual reality lessons not replace hands-on activities currently being used in classrooms; virtual reality should augment the existing methods. "VR (virtual reality) needs to be developed as an integral part of the educational and training process, implemented alongside other traditional and non-traditional tools."[4] The difficulty comes in designing classroom activities to take advantage of the power of virtual reality. Currently, no solid link has been established between a virtual reality exercise and a hands-on classroom activity, but this must be done for virtual reality simulations to be truly effective in the realm of education.

[2] Erenay/Hashemipour
[3] Dunn/Wardhani
[4] Erenay/Hashemipour

As a result of technological advances in the past couple of years, virtual environments have become extremely realistic, and children are captivated by the latest and most popular simulation games. Given the attraction of virtual reality, schools would benefit by joining the students' interest in virtual reality with educational software. Virtual environments allow students to engage in activities and scenarios without the physical and monetary constraints of the real world. Our project strives to integrate the virtual world in this learning process. Another use of the virtual simulation component is to move the testing phase of the Robolab program to the virtual world. Thus, once the students' Robolab program operates correctly in the simulation, they can then load the finished program into the RCX and run their fully functional robot. This would allow students to learn the robotics concepts of Robolab through both the simulated world and through physical construction. Additionally, the virtual reality application would expose them to concepts of simulation.

Another important advantage to this use of the virtual world is that students will be able to have long distance interactions with each other. Currently, when different classrooms share their Robolab programming, one class emails a Robolab file, known as a "virtual instrument" or VI, to another class; then this second class must load the Robolab program into their own RCX and either watch the robot in a different environment or recreate an environment identical to the one in the remote classroom. Virtual reality would allow students to email a Robolab file (VI) in which their robot and its actions are simulated in the same virtual environment. An example application of this sharing could be a class on the east coast sharing their programming with a class on the west coast. Students on the east coast could send students on the west coast specifications for how their Lego robot was constructed. The west coast class could test their own Robolab code on the east coast class's Lego robot construction. This would encourage interaction and experimentation between classrooms, as they could see which Lego robot constructions work best for specific tasks.

Background

Robolab was developed to create an intuitive way for students to program the Lego Mindstorms RCX. Robolab is based on National Instruments Labview, a program which provides a graphical development environment for control programming, data acquisition, and data analysis.

Students are able to program in Robolab by wiring together different icons that control Lego Mindstorms components. For example, the program in Figure 1 turns a motor on for two seconds. The icons wired together in the figure are known as Virtual Instruments (VIs), which are similar to functions in C or other traditional programming languages. A graphical development environment enables students without a firm grasp of reading/writing to develop their problem solving skills by engaging in programming tasks. This allows students to visualize high-level programming concepts, building skills along the way, without the slow learning curve of syntactical programming languages. Furthermore, Robolab can be tailored to different students' abilities with its progressive levels of complexity.

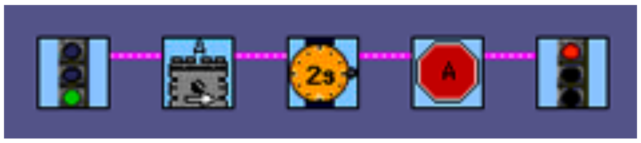


Figure 1: Robolab program to turn motor A on for two seconds

Last year, in our junior project, we were the first to incorporate virtual reality into Lego Mindstorms.[5] Students would create a program for their Lego robot in Robolab, and would simply substitute the regular stoplight marking the end of a program with a stoplight of our design. Instead of loading the control program into the RCX, the VI would output a VRML (Virtual Reality Modeling Language) file that showed a virtual vehicle controlled by their program. The students could then view that file in a web browser with a VRML plug-in. Once their program worked correctly on the virtual robot, the students could load the program into the RCX of a real Lego robot and watch it move in the real world. The simulation software thus provided a good method for students to alternate between using the simulation and the physical robots, so that they would gain hands-on experience with simulated *and* real objects. An added benefit is that it is easy for students in different locations to share the results of their Robolab programming since they would simply be able to swap VRML files.

[5] For more background on virtual reality education software in general, refer to our junior papers – http://www.princeton.edu/~ssubrama/JP.pdf
http://www.princeton.edu/~tbrower/Indie/VirtualRealityforLegoMindstorms.pdf

Last year, we accomplished a lot in terms of adding a virtual reality component to the Robolab development environment. However, there were a number of drawbacks – namely access, flexibility, and expandability. In order to view the VRML simulation file, it was necessary to download a VRML plug-in for use with a web browser. To complicate things, VRML files viewed with different plug-ins might be displayed differently and VRML plug-ins for Macintosh computers do not support all the functionality of plug-ins for Windows computers. Additionally, because our VI generated a VRML file to be viewed after the program's execution, there were no opportunities for real-time simulation features.

To fix some of these problems, over the summer, we integrated the Java External Authoring Interface (EAI) into our simulation to control the VRML display. With the Java EAI, we could get around the limitations inherent in using only VRML, enabling the simulation to become a much more powerful tool. With the Java EAI, it was possible to calculate the vehicle's movement in real time, allowing the simulation to handle much more complex tasks. Pre-computation imposed many limits on the possible tasks because of factors such as calculation time limits and file sizes. Using the Java EAI to control the VRML view also enabled us to utilize the full power of the Java programming language, including its extensive set of libraries, to further enhance the quality of the simulation; however, it presented a new problem – the Java EAI is not supported in Macintosh VRML viewers, so Macintosh computer users would not be able to use this software.

This year, National Instruments began development of a set of VIs, which enable VRML to be displayed within the Robolab environment. There were a number of tradeoffs in choosing whether to switch from the Java EAI to the Robolab VRML VIs. With the use of these new VIs, there was no longer any need for external application support such as web-browsers and VRML plug-ins, and all the elements of our simulation would be housed within Robolab itself. As a result, our simulation would be platform-independent, and all Robolab users could use it. However, the VRML VIs are currently in alpha stage development and a number of useful features have not yet been implemented. Among the unimplemented features are textures and extrusions, which are necessary for our simulation, so in any implementation of our simulation utilizing these VIs, we would have to work around these issues until updated versions of the VIs are released. Also, since the VRML VIs are in alpha development, they are not yet fully optimized for speed and rendering the scene would take longer than necessary. Despite these issues, we decided that the benefits of using these VIs were too great to be ignored and we decided to implement the simulation entirely in Robolab.

Approach

The main goals of our project this year were to fully integrate the simulation into Robolab and to improve the realism of the simulation. To integrate fully with Robolab, both programming and viewing of a simulation had to be done entirely within Robolab. This approach was possible this year due to the new VRML VIs under development. In order to make the simulation run within Robolab, we decided to use a scheme similar to that of using the Java EAI; we used the Robolab programming language and its extensive set of libraries to control the VRML display. This allowed us to port all the functionality of our Java EAI simulation package into the Robolab simulation, as well as take advantage of some Robolab specific functionality, such as image processing routines and user interface components such as dials, graphs, and array displays. As mentioned above, additional benefits from fully integrating the simulation into Robolab were obtaining platform-independence, and increasing ease of set up. Users no longer have to download a VRML plug-in and make sure it supports all the necessary features; they only need to install Robolab in order to run the simulation. In addition, the simulation is supported on all platforms which Robolab supports including Windows, Macintosh, and Unix machines.

To improve the realism and usefulness of the simulation, we added support for collision detection and handling along with fully functional light and push sensors, data logging, and environmental controls. In order to verify these added features, we challenged ourselves to support the real world task of programming a robot to find its way out of a maze. If our simulation could handle the programming of a robot finding its way out of a maze, our simulation would also be able to handle other programming tasks that require the same functionality and features. For example, with the new data logging features, we could use the robot's history of positions to map out a room; depending on how the robot moved, we would know where other objects are, such as walls or obstacles. Also, given that collision detection is possible between a robot and a stationary VRML object, it would be possible to expand the feature to handle collisions between different robots. This would allow for interaction between multiple robots in the same environment.

Methodology

The simulation was implemented using modular design principles in order to simplify its creation, testing, and maintenance. The main components of the simulation and their interactions are shown in the UML diagram found in Appendix A. The depicted hierarchy can be broken down in the three main groups:

1. **Input** VIs – obtain information from the user
2. **Calculation** VIs – determine what will happen in the simulation according to the user's inputs
3. **Visualization** VIs – display simulation data in a user friendly form

The input VIs include VRMLstoplight, primaryUserScreen, and executeSim. These three VIs gather information from the user about what will be simulated. The user must first create a Robolab program to control a vehicle, using the Robolab library of VIs. Then, instead of wiring their program into the normal Robolab stoplight, the user will wire their program into our VRMLstoplight VI. When the resulting Robolab program is run, the primaryUserScreen VI will pop up (see Appendix B, Figure 2 for a screenshot) in which the user makes choices about the type vehicle, environment, and sensors they want on their virtual vehicle (more about these choices in the "results" section below). After making their choices, the user clicks the "Run Simulation" button which calls the executeSim VI (see Appendix B for a screenshot) and starts the execution of the simulation. While the simulation is running, the user can change the lighting conditions in the virtual world as well as choose between different viewpoints to observe the vehicle.

The executeSim VI obtains user input during the simulation, but it is much more than just an input VI. ExecuteSim is the heart of the simulation. It parses through a user's control program, calling the calculation VIs and displaying the output of the visualization VIs. In order to understand how the simulation works, we will describe what goes on within the executeSim VI for the following Robolab control program:

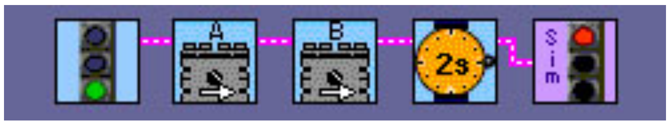


Figure 2: Robolab program to turn both motors on for two seconds

This simple program shown in figure 2 makes a vehicle go forward for two seconds. The information being sent into the stoplight is in the form of a cluster, which contains a Lego Assembly Language (LASM) version of the control program. Our simulation uses this LASM code to determine the movements of the virtual vehicle within the simulation. The process of translating the cluster of information into a visual simulation is diagrammed in the flowchart in figure 3.

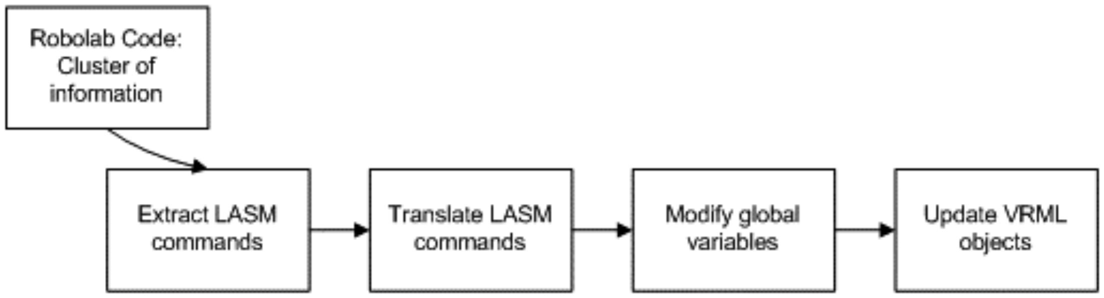
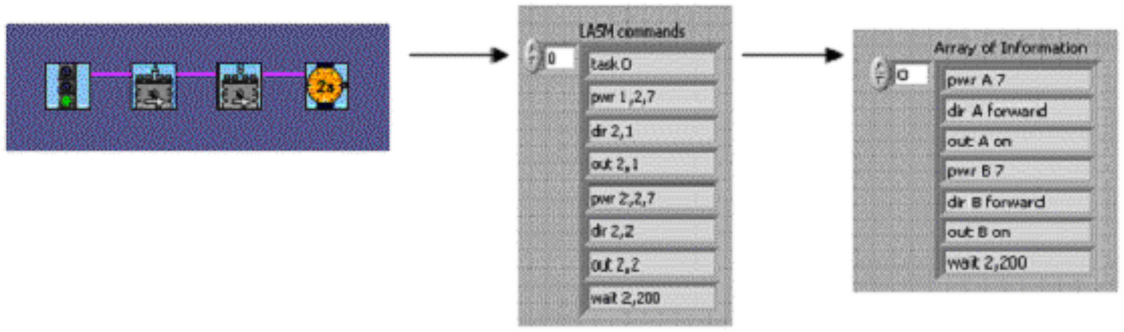


Figure 3: Flowchart of simulation events

This flowchart is a very simplified depiction of what is happening inside the executeSim VI. ExecuteSim takes as input the cluster of information from the user's Robolab program and then extracts the LASM commands from that cluster by calling the LASMRead VI. LASMRead goes through each of the LASM commands and translates them into a more readable form. For example, in the LASM command "dir 0,2", 0 represents motor A, and 2 represents forward, so the translated command would be "dir A forward". These translated commands are stored in a one dimensional array, which maintains the LASM command order. This conversion can be seen in Figure 4 below.


LASM commands

```
0
task 0
pwr 1,2,7
dir 2,1
out 2,1
pwr 2,2,7
dir 2,2
out 2,2
wait 2,200
```

Array of Information

```
0
pwr A 7
dir A forward
out A on
pwr B 7
dir B forward
out B on
wait 2,200
```

Figure 4: Picture chart of LASM translation

After the translation is completed, the LASMRead VI returns and passes the one-dimensional array of translated LASM commands to the executeSim VI. At this point, executeSim initializes all the global variables to their default values. The global variables keep track of the state of the vehicle, such as its position, rotation, and motor powers, along with several simulation variables, including the time and program counter.

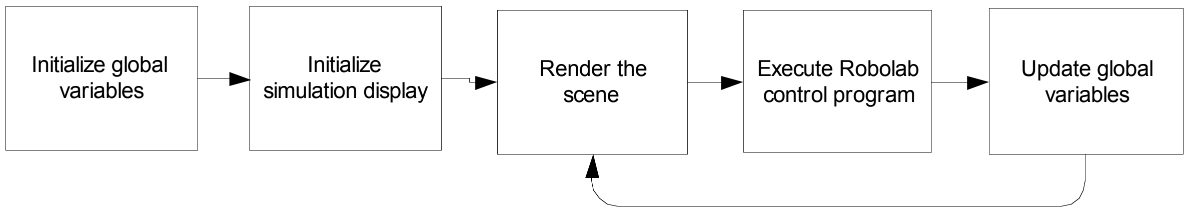

Initialize global variables
Initialize simulation display
Render the scene
Execute Robolab control program
Update global variables

Figure 5: Flowchart of ExecuteSim VI

As shown in Figure 5, after initializing the global variables, the simulation display is initialized to show the vehicle at its initial position in the virtual environment. The actual rendering of the scene is accomplished by using the refreshVRML VI, which contains calls to the VRML VIs we obtained from National Instruments. After completing all initializations, executeSim starts running the user's Robolab control program.

To run the control program, executeSim indexes into the LASM command array according to the value of the program counter, executes that command, and then changes the program counter accordingly (usually increments by 1). Then the command corresponding to the new program counter value is executed and this process continues while the value of the program counter is less than the size of the LASM command array. If the program counter is greater than or equal to the size of the array, the program is finished and the simulation ends. In our simple example program, the size of the LASM command array is 7, and there are no jumps, so the program counter will start at zero and be incremented 7 times before the simulation is completed.

The actual execution of each command is accomplished by calling subroutines which change the state of the vehicle and then update the VRML scene, if needed. We have written a subroutine for each LASM command the simulation supports. In our simple example program, only four out of the seven LASM commands that we support are used: pwr, dir, out, and wait. These commands are handled by the setPower, setDir, setOut, and rcxWait subroutines, respectively. SetPower and setDir are used to calculate the velocity that each wheel will be moving. A positive velocity means that the wheel will be moving forward and a negative velocity means the wheel is moving backward. The relationship between wheel velocity and motor power setting was obtained experimentally by calculating a real Lego vehicle's speed at different motor settings and then fitting a curve to the data.

SetOut calculates the vehicle's motion as a function of the two wheel velocities. To do this, we simplified our vehicle model to be the axle connecting the wheels. The wheel velocities impart a rotational velocity on the axle, and the two wheels rotate with equal rotational velocity about some point, which is called the axis of rotation. Using the equation for solid body rotation, w = v/r, where w is the angular velocity, v is the tangential velocity, and r is the distance to the axis of rotation, we are able to calculate the distance from each wheel to the axis of rotation and, as a result, the position of the axis of rotation. There are two different equations for finding the distance from a wheel to the axis of rotation, depending on whether the axis of rotation will be between the wheels. The two possible cases are shown below. For the case (case 1 of figure) where the axis of rotation is to the left or the right of the vehicle, the equation is $r_1 = \frac{v_1 * l}{v_2 - v_1}$. For the case (case 2 of figure) where the axis of rotation is between the two wheels, the equation is $r_1 = \frac{v_1 * l}{v_1 + v_2}$.

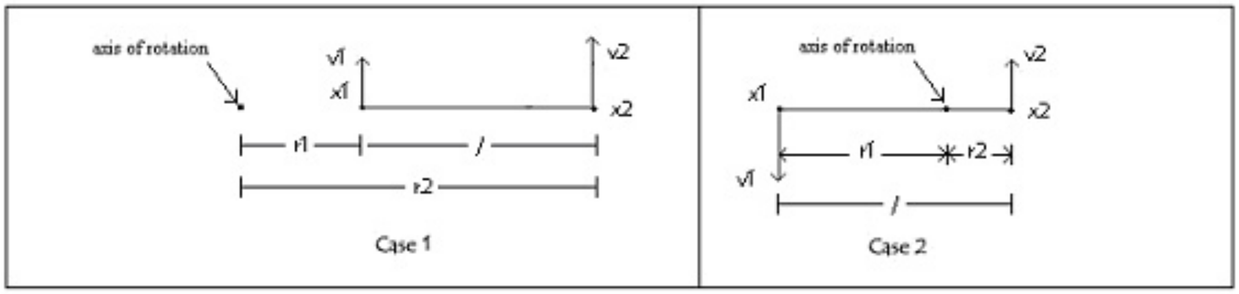


Figure 6: Diagrams of vehicle wheels with respect to axis of rotation

Knowing the axis of rotation, distance to the vehicle, and the angular velocity of the vehicle, it is possible calculate how the vehicle will move as time evolves. SetOut calculates these values, according to the wheel velocities at the time of the calculation, and stores them in global variables for later use. It should be noted that setPower, setDir, and setOut do not result in vehicle movement. These VIs merely set global variables, changing the internal state of the vehicle. Vehicle movement does not take place until the execution of a LASM "wait" command. This process mimics the execution process of the Lego RCX; if the Robolab control program does not contain a "wait" statement, the vehicle will not move (or it will render the RCX unusable until a reset).

The rcxWait VI is responsible for moving the vehicle by updating the state variables for position and rotation, along with refreshing the VRML scene of the simulation. The VRML scene is refreshed every 0.08 seconds (13 times a second) because of time constraints imposed by program execution time. It was not possible to achieve a faster refresh rate because it took around 0.08 seconds to do all the calculations associated with moving the vehicle and updating the VRML scene. For the most part, this slow execution time is a result of using the beta version VRML VIs from National Instruments which have not been optimized for speed. Once these VRML VIs are optimized, better results will be possible.

Robolab programming essentially works by telling motors to run at certain speeds for certain amounts of time. As a result, the accuracy of timing is critical to the accuracy of the simulation; the rcxWait VI is designed with this fact in mind. It is easiest to understand rcxWait as a loop that executes once every time the VRML scene is refreshed, for a given duration of time. In our example program, the command is to wait for two seconds, so in the simulation, the loop within rcxWait will execute every 0.08 seconds for 2 seconds. Each loop execution accomplishes three tasks, besides keeping track of time: it calculates the new position and orientation of the vehicle, updates the vehicle's state variables to reflect those changes, and then updates the VRML scene with the new state variables. It is a straightforward process to calculate the new position and orientation of the vehicle given the position of the axis of rotation, the distance of the axis of rotation to the vehicle, the angular velocity of the vehicle with respect to the axis of rotation, and the amount of time elapsed since the last calculation. Each operation takes a set amount of time, independent of processor speed; thus, all computers will show the same simulation (as long as the minimum speed requirements are met).

The above procedure can handle any Robolab code that moves the vehicle in any direction with no obstacles. However, if the user has selected an environment in which the vehicle may collide with walls, then we need to perform collision detection calculations. For each possible maze environment, we created a corresponding bitmap file that has pixel values of '1' wherever a wall is present in the simulation environment and '0' everywhere else in the bitmap file. There is a similar bitmap file for the vehicle matrix. We read in from these bitmap files to create an environment matrix of 1's and 0's (1's where the black lines of the maze are) and a vehicle matrix of 1's and 0's (1's where the vehicle lies). The vehicle matrix size is just a little larger than the actual vehicle so that the vehicle can rotate within the matrix frame without crossing over the matrix boundary. To calculate for collision, we use these two matrices. When the vehicle moves, the vehicle matrix is rotated accordingly. We use the vehicle's position in the simulation to index into the appropriate area in the environment matrix, and copy out a sub-matrix that is the same size as the vehicle matrix. We then perform a matrix AND between the vehicle and the sub-matrix that we have just obtained from the environment matrix. If there are any 1's in this resulting matrix, then we know that there is a collision because two 1's within the matrices overlapped. For example, in the figure below, we can see that the environment sub-matrix (See below figure, left) consists of a vertical wall. The vehicle (below, middle) is approaching this wall. Based on the resulting matrix (below, right), we can see that the vehicle did in fact collide with the wall.

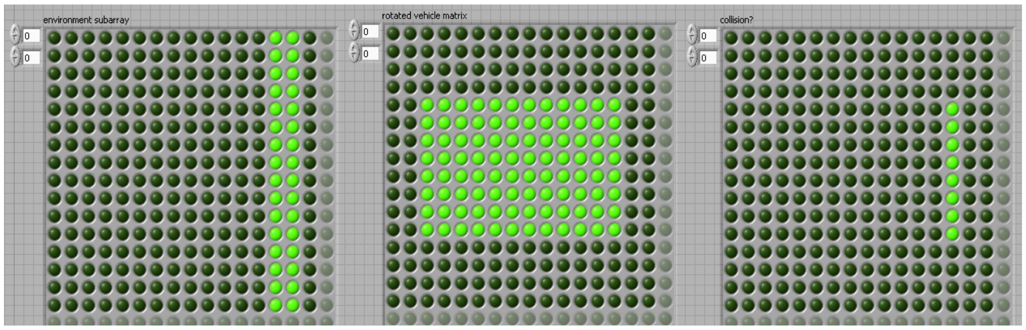


Figure 7: Collision detection matrices
From left to right: environment sub-matrix, vehicle matrix, and resulting collision matrix

If there is a collision, as in this example, we call a separate collision handling VI which tries to realistically simulate the effect of the collision. Our collision handler determines where on the vehicle the collision took place. More specifically, if either the front left or back right corner of the vehicle hit the wall and the vehicle was rotating clockwise, we can assume that the vehicle was turning away from the wall as it collided, and will continue to turn away. If instead, the vehicle was turning counter clockwise (and the same corner was hit), the vehicle would keep trying to turn into the wall and get stuck. The reverse scenarios apply for the opposite corners – front right and back left. To determine where the vehicle hit the wall, we use two matrices that are the same size as the vehicle matrix, but store only small cluster of points around the corners of the vehicle; one matrix is for the upper left and the lower right corner clusters of the vehicle, and the second is for the opposite two corners. We can now apply the same approach that we used in detecting collision by matrix-ANDing each of these two matrices with the environment sub-matrix that we previously calculated. If there is a resulting collision with both of these operations, we know that an entire side of the vehicle hit the wall, and the vehicle will stop. For example, if the vehicle is moving in a straight path and collides head-on with a wall, both corner matrices would have collisions with the environment sub-matrix, and we would simply stop the vehicle. If not, then we determine which corner matrix had the collision, and react to the collision based on the vehicle's rotation. For example, if the vehicle is curving towards the right, and its front left side collides with a wall, we assume slippage and we rotate the vehicle by an amount that is dependent on the speed of the vehicle. The faster the vehicle is going, the faster it needs to turn to keep it from crashing through the wall. To calculate the amount the vehicle should rotate, we first take the average speed of the two wheels and then multiply it by 10 degrees divided by the number of motor power levels. Thus, if both motors on the vehicle were set to 7 (the highest power), then the vehicle would rotate by 10 degrees. The vehicle will continue to collide until it rotates to be parallel to the wall so that it can move away from the wall.

The sensor detection works very much like collision detection. For push sensors, we use the same notion of performing a matrix AND to see where a push sensor collides with a wall. The main difference is that the vehicle matrix only has 1's where the push sensors are located instead of over the entire object. The push sensor performs exactly like a real push sensor in that it allows a user's control program to react to the vehicle colliding with an object.

The light sensor has two different scenarios: pointing parallel to the ground or looking down at the ground. When the light sensor is pointing ahead, we take a snapshot of what the sensor would be seeing from its viewpoint, and we then take the average pixel intensity value from an area in the middle of this picture. Unfortunately, VRML does not support attenuation of light reflecting off a surface (though attenuation from a light source is in the VRML 2.0 specification, it is not supported by the current VRML VIs). In other words, as we get closer to a wall, there is no change in the light intensity; correspondingly, there is no change in the brightness value. To account for this, we multiply the brightness value by a scaling factor of $1/r^2$, where r represents the distance from the light sensor to the closest wall in its vision. If this average intensity reaches the desired brightness value (a default value of 55, or a value the user provided in the Robolab code), we perform the action that the user specified.

The current Robolab does not support textures, so we cannot create a ground texture with different brightness. Instead, we use a VRML object to represent the ground. We then create a mask at the ground level, and place the bitmap image underneath so that it appears as if the vehicle is moving on top of this textured ground. When the vehicle moves, we sub-index into a matrix representation of the bitmap to find the brightness value of the current pixel. Then, we perform the corresponding action that the user programmed.

User Interface

The Robolab front panel (see Appendix B, Figure 1) is made up of a view of the simulation, a speedometer, an odometer, an angular velocity meter, a brightness meter, a history of positions graph, lighting controls, and two user selection menus for viewpoint and environments choices. The user can select from different pre-made environments (currently one with a maze and one without), which they must decide upon before the simulation is run. We also created five main viewpoints (top, front, left side, right side, and back) from which the user can view the simulation. The user can change the viewpoints or lighting controls at anytime before or during the simulation.

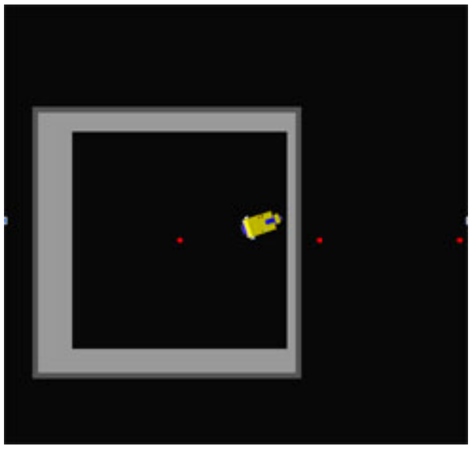


Figure 8: Simulation window

The data logging mentioned earlier consists of these different meters and graphs that display information about the vehicle's movement for the user. All of the data logging controls and the main simulation window seen in the above panel are based on global variables. The simulation window (see figure 8) is controlled by the array that stores all the VRML objects. The angular velocity is calculated each time the out command is called, which is when a motor is turned on or changes speeds. The global variable that stores this variable feeds directly to the display above so that the meter changes accordingly with each calculation. The speedometer is updated when the wait command is reached. The speed, similar to the angular velocity, is computed and stored in a global variable that feeds directly into the speedometer dial above. There is yet another global variable to calculate and store the distance the vehicle has traveled, displayed in the odometer.

The locations that the vehicle travels are stored in two global variables, x and y, representing the vehicle's coordinates within the environment. Robolab has a user interface such that we could hook up these two global variables to a histogram graph and it plots and connects the points that the vehicle travels (see Figure 9). The brightness meter is also connected to the variable we use in the program to determine the light intensity that the light sensor is reading at a certain point.

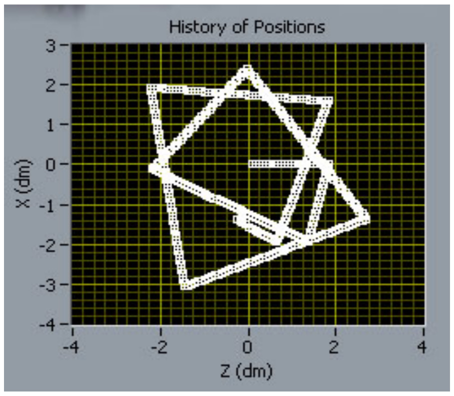


Figure 9: History of positions

Results

The result of our development efforts is a library of VIs with supporting materials that enable the simulation of a Lego Mindstorms vehicle in different environments. The virtual vehicle moves accurately with any combination of motor powers and directions. In addition to these basic movements, the virtual vehicle is capable of using a push sensor and/or a light sensor to explore and interact with the virtual environment. In terms of Robolab programming, the simulation can handle 50 Robolab programming VIs, including all the VIs in Inventor level 3, other than task splits, forks, sounds, lights, and the loop structure.

To use the simulation, our Simulation Stoplight VI must be substituted for the regular stoplight that ends a Robolab program. When the program with the simulation stoplight is run, a front panel pops up in which the user needs to make choices about the nature of the simulation they would like to run. A screenshot of the front panel can be seen in Appendix B. The user is required to choose a maze (different placements of walls), a vehicle, and a ground image. Currently there are two very simple maze choices: maze 1 is an 80cm x 80cm room and maze 2 contains no walls. There is only one vehicle choice, and the user has a choice between two ground images: a swirly pattern and a gradient pattern. Even though there are currently very limited choices, we set up an infrastructure that makes it easy to add more choices when they become available.

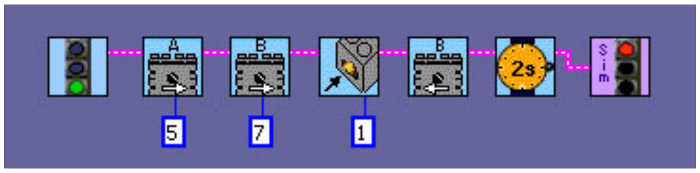


Figure 10: Robolab program that demonstrates use of push sensor

The sensor choice options apply only if the program being simulated contains sensors. If the user's Robolab program is designed to use a push sensor, like the program in Figure 10, they have to set up a virtual push sensor.

A virtual push sensor is defined by the matrix controller in the options front panel (shown in Figure 11). When the front panel pops up, the matrix controller contains the outline of the virtual vehicle. The user can define their virtual push sensor by selecting points outside of the vehicle outline which will register a collision with an external object and set off the push sensor. If points inside the vehicle outline are selected, the virtual push sensor will have no effect.

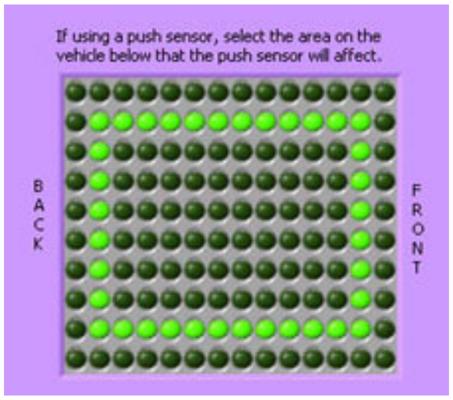

If using a push sensor, select the area on the vehicle below that the push sensor will affect.
BACK
FRONT

Figure 11: User selection for placement of push sensor

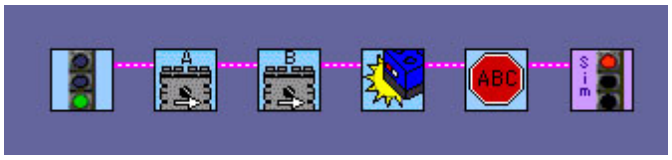


Figure 12: Robolab program that demonstrates use of light sensor

If the user's Robolab program is designed to use a light sensor, like the program in Figure 12, they have to set up a virtual light sensor located at the front of the vehicle.

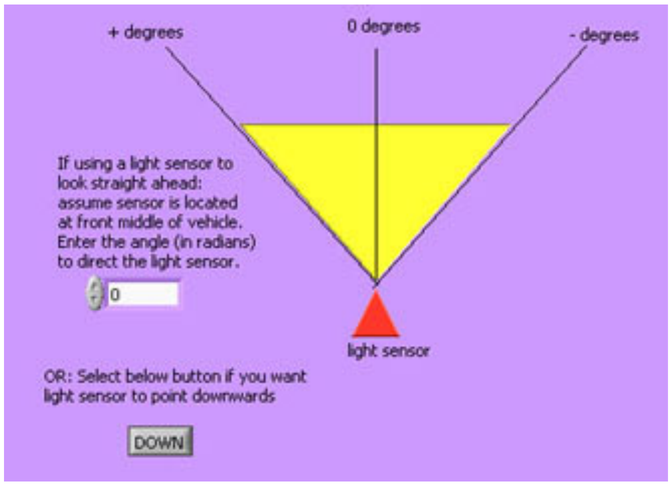

+ degrees
0 degrees
- degrees
If using a light sensor to look straight ahead: assume sensor is located at front middle of vehicle. Enter the angle (in radians) to direct the light sensor.
0
light sensor
OR: Select below button if you want light sensor to point downwards
DOWN

Figure 13: User selections for light sensor

To set up a light sensor, the first decision the user must make is whether it will be looking parallel to the ground or looking down at the ground (see Figure 13). Looking parallel to the ground is useful for navigating a maze or telling how far the vehicle is away from an object. Looking down at the ground is useful for following a line on the ground or doing things based on the darkness of a ground pattern. If the light sensor is pointing parallel to the ground, the user can also specify what angle the sensor points.

After setting the appropriate simulation options, the user presses the green “Run Simulation” button to start the simulation. As previously mentioned, the simulation is capable of both detecting and handling collisions between the vehicle and the environment. The accuracy of the collision detection is one-eighth the width of the RCX which is 0.08 VRML units which translates to 8 millimeters in the real world. This means that a collision will be detected when the edge of the vehicle is within 8 millimeters of a wall. In simulations with the light sensor pointing at the ground, we were able to reliably detect a line 2 centimeters wide at motor powers of 7. With a line thinner than 2 centimeters, the light sensor did not reliably sense it.

Discussion

There are limitations inherent in all simulations. A simulation can never be exactly the same as the real thing. Simulations make use of models of the real world that cannot hold under all conditions. No matter how good a model is, it can always be broken because models are based on measurements of real world data and estimations of real world phenomena. No sensor is perfect and so no data is completely error free. In spite of these facts, simulations are still useful tools because they give us a good idea of what will happen given certain assumptions.

Though our model of the Lego vehicle is a good estimation of how a real one actually behaves, it is not perfect for all cases. However, instead of this inaccuracy being a major weakness of the simulation, it actually serves a useful purpose. By comparing the simulation’s results to the actions of a real Lego vehicle, students can learn about both the strengths and the weaknesses of simulations. That said, we have done as much as possible to ensure the accuracy of the simulation. We tested many combinations of motor powers and direction with our simulation and with a real vehicle to see how well they matched up. For detailed diagrams of the test cases and vehicle motions, see Appendix C. In all cases, the simulated and actual results matched reasonably well.

The above test showed us that the simulation performed well in the most basic cases of movement. To further test the accuracy of the simulation, we tested the more complicated scenario of a line following vehicle. The Robolab control program for this scenario is shown below.

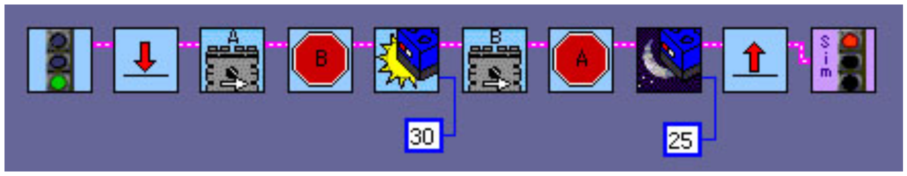


Figure 14: Line-following Robolab program

We first used our simulation stoplight to simulate the line following robot in an environment with no walls and a swirly pattern on the ground. We recorded the history of positions for a 1 meter by 1 meter area around the starting point of the vehicle. Then we set up the same scenario in the real world by creating a 1 meter by 1 meter print out of the same swirly pattern, to scale with the simulation. It should be noted that it took a very long time to create this 1 meter by 1 meter environment for the vehicle to interact with and creating the entire 4 meter by 4 meter space that is available in our simulation would take an incredible amount of time, unless better tools were available (we used a laser printer to print out sheets of paper with parts of the swirly pattern and then put them together with tape like a puzzle, it required 30 sheets of paper). After creating our swirly pattern, we put it on the ground and hung a webcam directly above it to capture the motion of the vehicle. We then loaded the program into the RCX of the real vehicle and ran it in our environment, recording the progress of the vehicle. Finally, we analyzed the video, and extracted the path of the vehicle by measuring the position of the vehicle every 5 video frames (video was running at 15 frames/sec). We corrected for the distortion of the picture resulting from very cheap camera optics by using elementary image transformations and plotted the results on the same axis as the history of positions from our simulation. The resulting plot can be seen in figure 15.

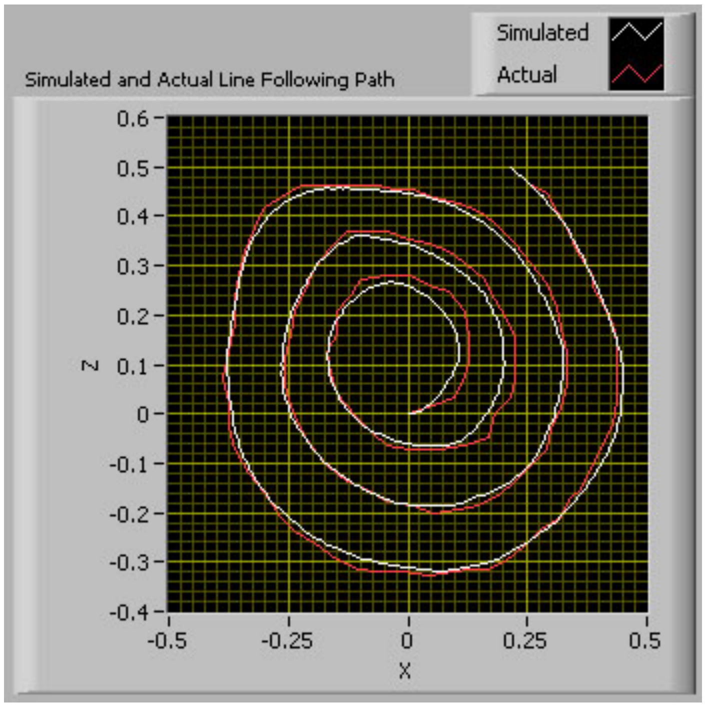


Figure 15: Simulated vs. Actual line-following graph

The simulated and actual results matched very well. The discrepancy in positions in the middle of the plot can be partially attributed to the fact that the camera was shaking slightly at the start of the test.[6] These results show that the simulation's line following has a high degree of accuracy and works well. Due to time constraints, we were not able to build the other environments and perform similar tests for accuracy. However, through observations we know that the other features perform reasonably accurately.

One feature that does need more work is collision handling. Our simplified collision handling is based on observations of a physical robot's behavior when colliding with a wall. We made many simplifications in order to speed up and simplify the collision handling routine. Therefore it is not exact; however, it provides a fairly good representation of how a vehicle would behave in the real world. The reaction of a vehicle collision in the real world is dependent on many factors, including friction, surface material, battery power, slippage, etc. Most of these factors are currently not feasible to model due to their complex nature; therefore, we simplified the problem by assuming a constant value for many of these factors. These simplifications also limits the accuracy of the simulation in general since the real vehicle will react to changes in these conditions.

[6] The video can be viewed at http://www.princeton.edu/~tbrower/lilSniffy.mov (requires QuickTime)

Another limitation of our simulation (and simulations in general) is the finite amount of choices for parameters within the simulation. Our simulation currently restricts users to one, standard vehicle with at most two sensors (one light sensor and one push sensor). Furthermore, the use of these sensors are limited in their placements. The vehicle is also limited in that the motors are positioned on the vehicle in a specific way and there can be only two motors on the vehicle, connected directly to wheels. Aside from the limitations in the vehicle, the simulation does not yet support all Robolab commands. However, these limitations are not necessarily a bad thing. They could limit the ways in which a student can solve a problem, but the limitations could also be used to force a student to concentrate on a specific task, such as line following algorithms, instead of just playing with the Lego bricks. In the end, the usefulness of the simulation depends on how it is utilized. If one tries to do too much, it will not work, but if one tries to do too little, it will be a waste. Teachers and students must find the happy medium where the simulation can have the greatest benefit.

Some of the limitations of the simulation are the results of limitations in the tools we used to create it. Specifically, the version of National Instruments VRML VIs imposed a number of performance limitations on our program development. As mentioned before, the VRML VIs are only in alpha development and so they are not optimized, contain some bugs, and are do not fully support the VRML 2.0 specification. No optimization means that the VIs are running a lot slower than they have to and this limits the performance of our simulation since we have a limited amount of time in which to do calculations in the simulation's loop. On top of this, texture are not supported so we had to create "fake" textures for the ground by making the ground a certain color, masking that color out, and replacing it with a scaled version of the ground image. This step added a lot more calculations to the program loop, which was already overburdened by the slow VRML VIs. Furthermore, we were only able to do this masking for one of the four viewpoints available to the user. So if you try to use a viewpoint with no masking, there will be no ground image, but the simulation will run significantly faster. Once a release version of the VRML VIs come out, our simulation will enjoy vast performance improvements since these extra steps and running time requirements will not be necessary.

Conclusion

The goal for this project was to use Robolab to create a prototype for software to simulate the movements of a Lego robot in the virtual world. We accomplished this goal by integrating the simulation into Robolab itself by using the new VRML VIs currently in alpha stage development. By integrating the simulation into Robolab, we achieved platform independence and eliminated the need for external software support. In order to develop a simulation package ready for commercial release, several components of the simulation must be improved. The development of improved components can occur rapidly and independently of each other due to the simulation's modular design. These improvements are chiefly dependent upon obtaining a beta or distribution release of National Instrument's VRML VIs. When these are available, the speed of the simulation will be dramatically improved and as a result, the timing of the simulation will become more accurate.

To further improve our simulation, beyond integrating a release version of the VRML VIs, a lot can be done. To improve the simulation's functionality, we can expand it to support all Robolab programming VIs including tasks, forks, loops, containers, sounds, etc. We can also add more maze and ground choices so that users can explore different environments with the virtual vehicle. The collision handling VI could also be upgraded to a better approximation of vehicle collision dynamics. In addition, we could add support to allow more than one vehicle to interact within the same environment, along with adding support for multiple sensors on each vehicle. It would also be worthwhile to take more parameters into account in the model of the vehicle, such as battery power and different wheel sizes.

Virtual reality simulations are an important tool that should be integrated into the curriculum of elementary and high schools, as well as universities. Our simulation provides an excellent way to introduce this subject to any grade level. Class work utilizing both our simulation and the Lego Mindstorms Robotics Kit will provide a concrete link between a simulation and a hands-on classroom activity. Through this link, students will be able to learn about the benefits and potential shortfalls of utilizing simulations to aid in designing solutions for real world problems. The integration of simulations into elementary and high school curriculum will better prepare students for their future, no matter what field of study they decide to pursue.

References

1. Brower, Theo. “Virtual Reality for Lego Mindstorms”. Princeton, NJ 2003.

2. Dunn, Tim L. and Wardhani, Dr. Aster, A 3d Robot Simulation for Education, Melbourne, Australia, 2003.

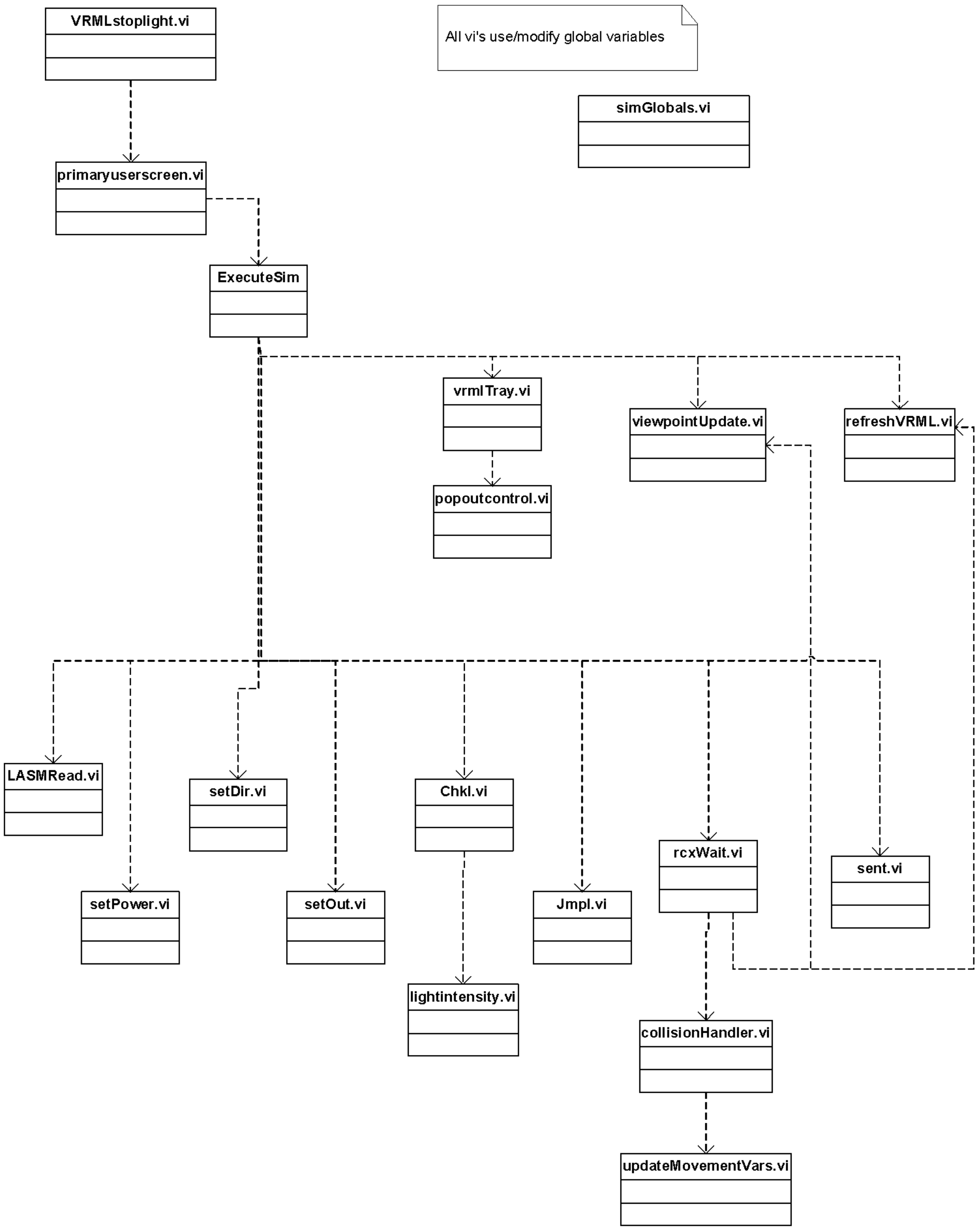
3. Erenay, Ozan and Hashemipour, Majid, “Virtual Reality In Engineering Education: A CIM Case Study” http://www.tojet.sakarya.edu.tr/current%20articles/ozan.htm

4. Portsmore, Merredith, “ROBOLAB: Intuitive Robotic Programming Software to Support Life Long Learning”, APPLE Learning Technology Review, Spring/Summer 1999. http://www.ceeo.tufts.edu/robolabatceeo/resources/articles/default.asp

5. Subramanian, Sushmita. “Virtual Reality for Lego Mindstorms”. Princeton, NJ 2003.

Appendix A

This UML diagram is a simplified illustration of how the simulation is organized.



Appendix B

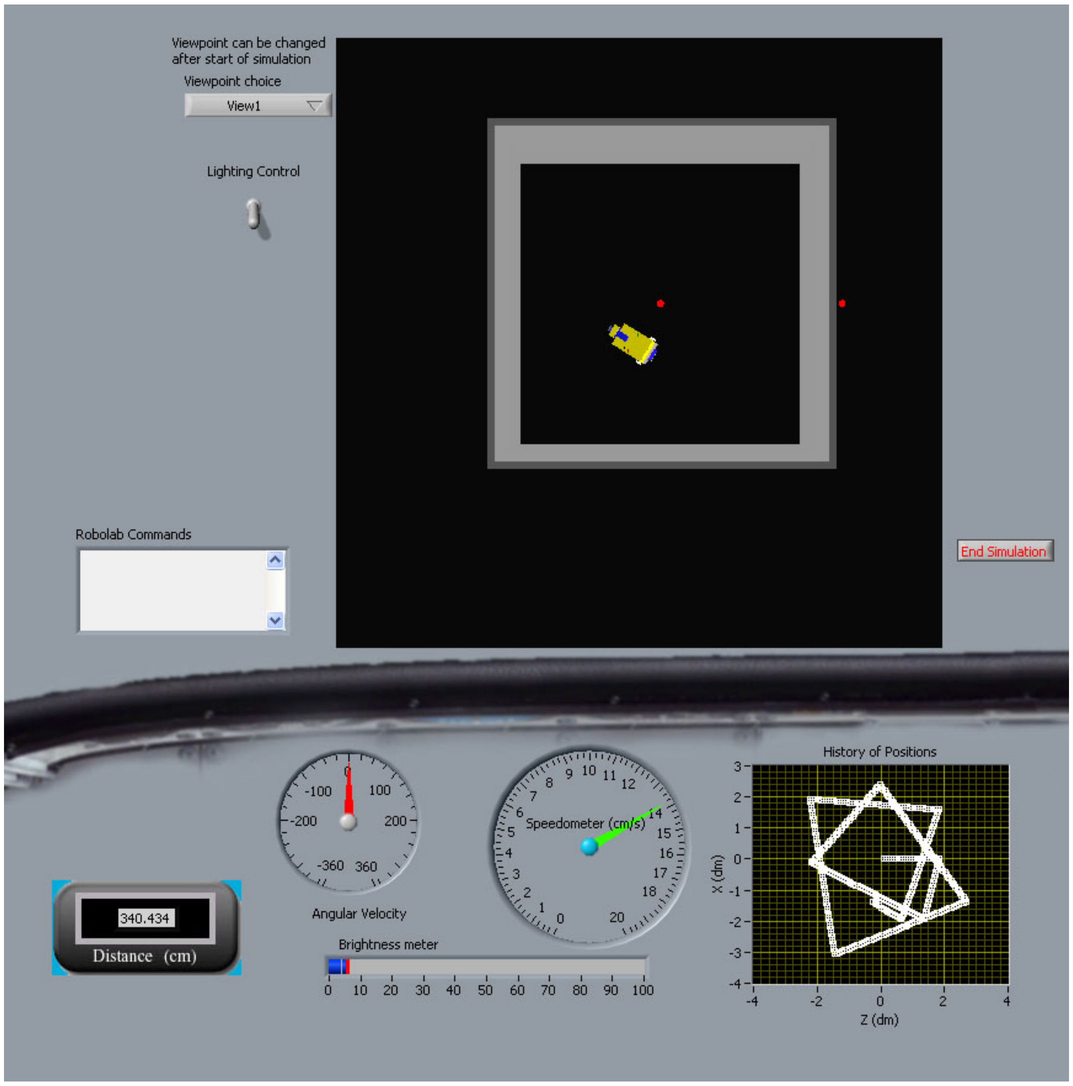
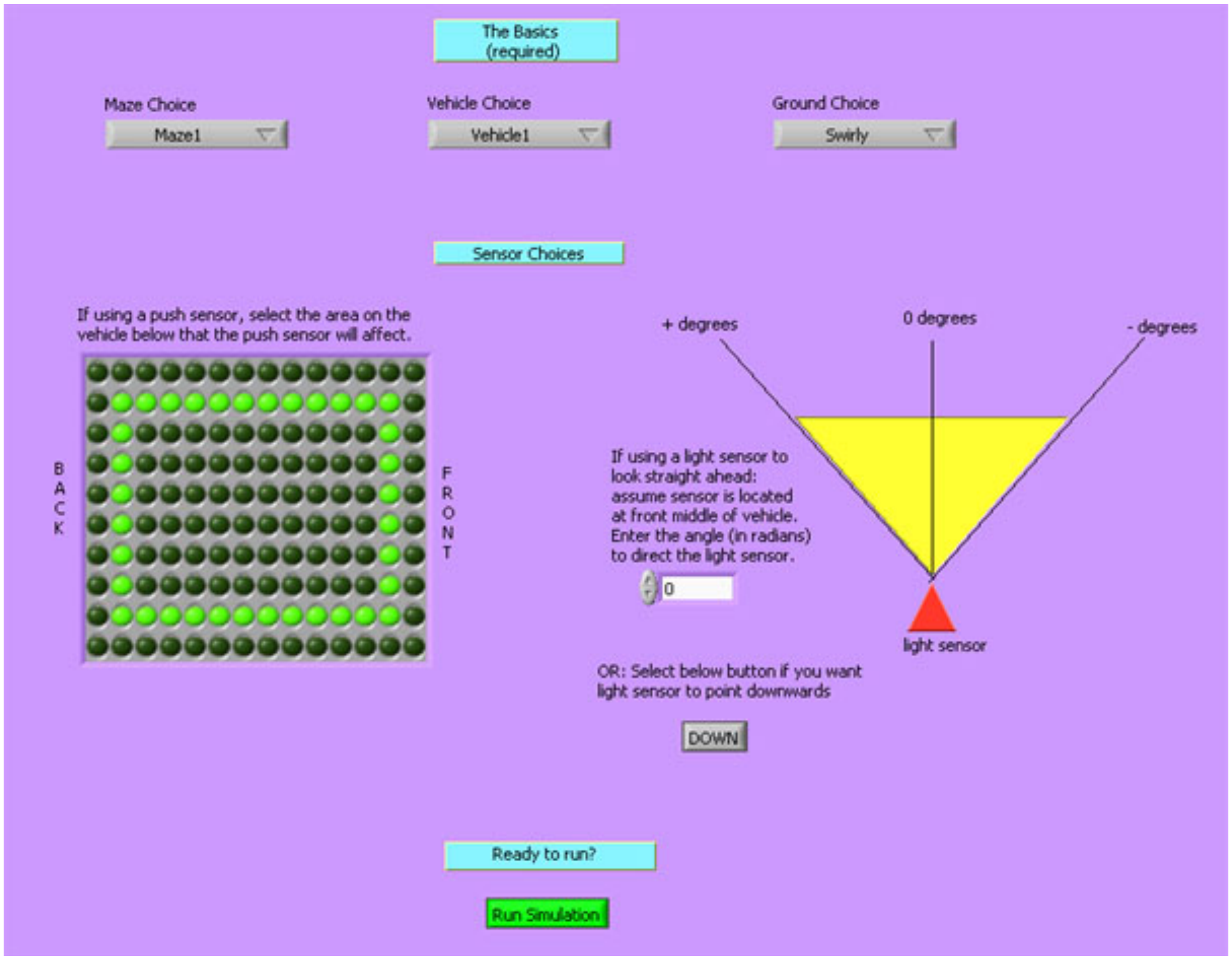


Figure 1: Screenshot of Robolab front panel


The Basics (required)

Maze Choice: Maze1

Vehicle Choice: Vehicle1

Ground Choice: Swirly

Sensor Choices

If using a push sensor, select the area on the vehicle below that the push sensor will affect.

If using a light sensor to look straight ahead: assume sensor is located at front middle of vehicle. Enter the angle (in radians) to direct the light sensor.

0

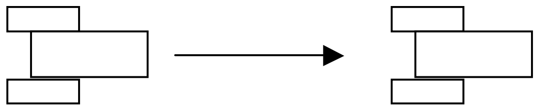
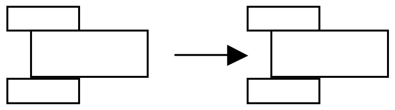
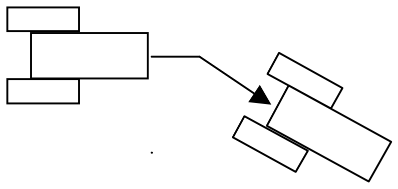
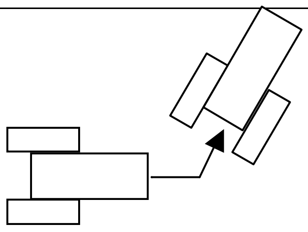
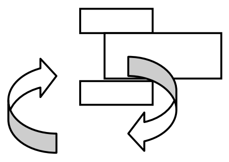
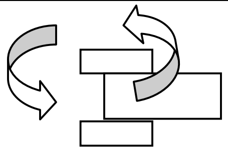
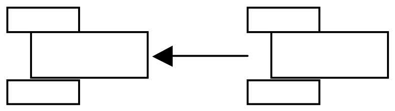
OR: Select below button if you want light sensor to point downwards

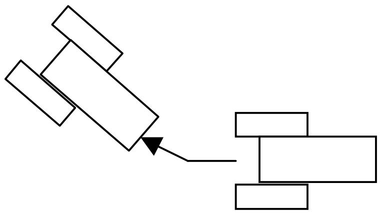
Ready to run?

Figure 2: Screenshot of User Selection Menu

Appendix C

The arrows in the diagrams below indicate the motion of the vehicle, given motor powers and directions for motors A and B.

Motor A (power, direction)	**Motor B (power, direction)**	**Results**
7, forwards	7, forwards	
4, forwards	4, forwards	
5, forwards	2, forwards	
1, forwards	7, forwards	
4, forwards	4, backwards	
4, backwards	4, forwards	
5, backwards	5, backwards	

1, backwards	5, backwards	
6, backwards	2, backwards