User's Guide to Thor's Hammer
A Thymio II Simulator and Visualizer

Alex Brodsky

July 21, 2017

Abstract

Do you want to use Aseba Studio to program a Thymio II? Except, you don't have a Thymio II with you. This used to be a problem because Aseba Studio does not allow you to program in disconnected mode. But, no more. Now, if you do not have a Thymio II, you can use Thor's Hammer, a simulation and visualization system for the Thymio II robot. The system consists of two programs: A simulator (*Thor*) and a visualizer (*Hammer*). *Thor*, which is short for Thymio on Remote, simulates a Thymio II robot. *Hammer* allows you, the user, to visualize what your Thymio is doing, create a world (environment) in which the Thymio runs, and interact with the Thymio, by pressing its buttons and tapping it.

Note: Hammer is still in the development stage, so it may crash. Be sure to save your work often. Send bug reports to `abrodsky@cs.dal.ca`.

Contents

Chapter 1

Getting Started

1.1 System Requirements

Ensure that Aseba Studio (or Studio for Thymio II) is installed (`https://www.thymio.org`). As well as an up to date Java Runtime Environment (JRE). Use the *Hammer* visualizer by connecting to a remote *Thor* server via the Internet (the default), or by installing a local *Thor* server.

1.2 Quick Install

Download the `hammer.jar` (`https://www.cs.dal.ca/~abrodsky/other/Hammer.jar`) and store it in a convenient place such as the Desktop or Applications folder.

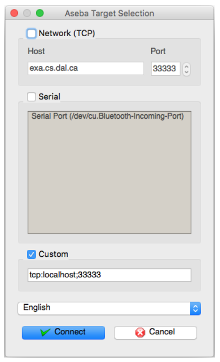
1.3 Running Hammer

Run *Hammer* by clicking on the `Hammer.jar` file. The user interface depicted in Figure 1.1 should appear.

1. If needed, change the *Thor* server field in the Connection Panel (top right in *Hammer*'s main window) to the desired server. The default setting `exa.cs.dal.ca` should work.
2. Run Aseba Studio (or Thymio II Studio). When the connection dialog box (on the right) appears, check the **Custom** option in the dialog. Ensure the connection target is:

   ```
   tcp:localhost;33333
   ```

   which is typically the default, and click on **Connect**.


Aseba Target Selection
Network (TCP)
Host: exa.cs.dal.ca
Port: 33333
Serial
Serial Port (/dev/cu.Bluetooth-Incoming-Port)
✓ Custom
tcp:localhost;33333
English

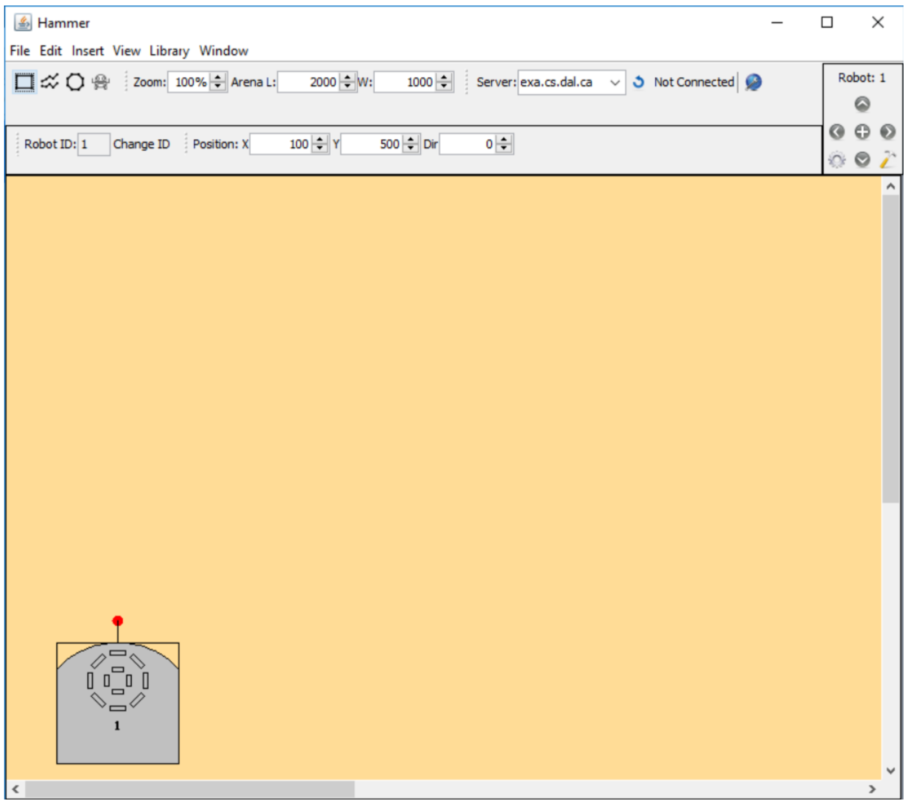


Figure 1.1: The visualizer's user interface.

3. If the connection fails because the Thor simulator is not responding or the network is down:
   (a) Switch back to the Hammer application.
   (b) Ensure that the Internet connection is working and that the correct server is specified in the *Server* field of the *Arena toolbar*.
   (c) Click the **Reset** button, which is located in the *Arena toolbar*, to the right of the *Connection Status* field.
   (d) Switch back to Aseba Studio, and see if the problem is resolved.
4. That's it! Enter a program into Aseba Studio to run on a simulated Thymio II and use *Hammer* to visualize and manipulate the simulation.

Error and information messages will be displayed in the *Log window* in the bottom right corner of the main window. All visual Thymio actions can be viewed in the arena (the yellow panel). Write a simple program to make the Thymio move, run it, and see what happens!

Note: Hammer is still in the development stage, so it may crash. Be sure to save your work often. Send bug reports to `abrodsky@cs.dal.ca`.

Chapter 2

Installation

This chapter covers the installation of the *Hammer* visualizer and the *Thor* simulator.

2.1 Installing the Hammer Visualizer

1. Ensure that a current version of the Java Runtime (JRE) is installed.

2. Download `Hammer.jar` at `https://www.cs.dal.ca/~abrodsky/other/Hammer.jar` and store it in a convenient place such as the Desktop or Applications folder.

3. Run *Hammer* by double clicking on the `Hammer.jar` file. If working on a Mac and not permitted to do this, then:

   (a) Run the `Terminal` program in the `Utilities` folder in the `Applications` folder.

   (b) Enter the command:

   ```
   java -jar Desktop/Hammer.jar
   ```

   This should run the *Hammer*. The user interface is depicted in Figure 1.1.

2.2 Installing and Running the Thor VM (optional)

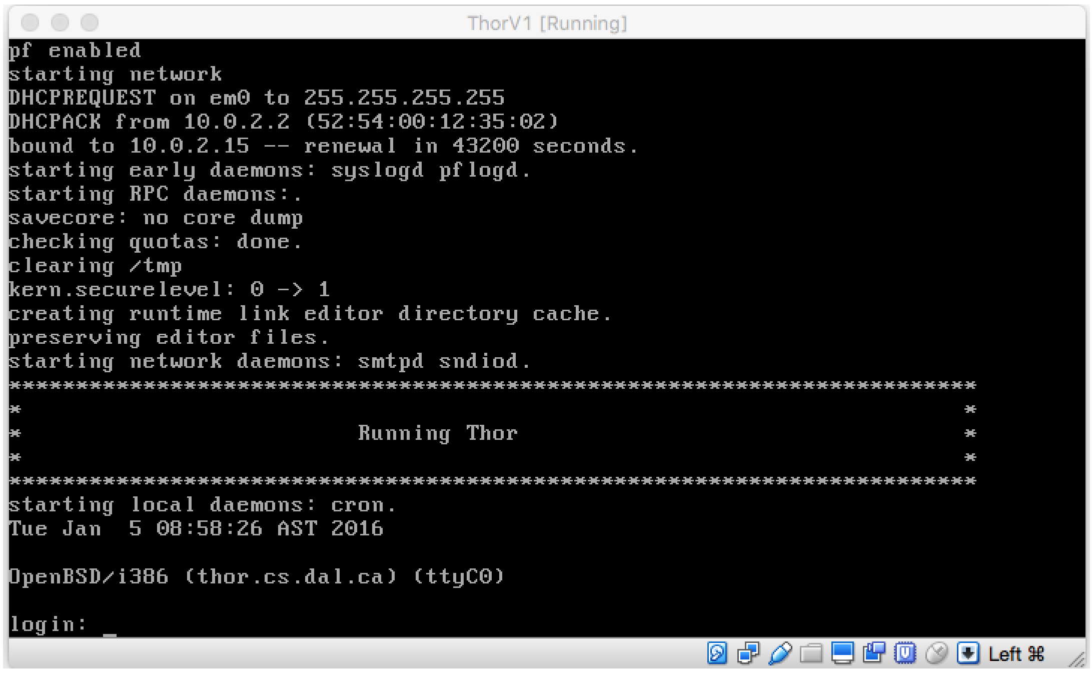
Running a local *Thor* server will yield better performance than using the default remote server. In an ideal world, a version of *Thor* would be available for every type of system. However, in this world, there are a plethora of different operating systems and configurations. Providing an executable for each of the systems is not sustainable. Instead there is a single guest Virtual Box image that will run the server on any platform that supports Virtual Box. While the install process is slightly more involved, it is a sustainable way to make available a local *Thor* server.

2.2.1 Installing the Virtual Machine

1. Download and install Virtual Box from `https://www.virtualbox.org`.
2. Download the *Thor* guest image (`ThorV?.zip`) `https://www.cs.dal.ca/~abrodsky/other/ThorV4.zip` Note: the `?` in `ThorV?.zip` is the version number.
3. Unzip the archive and place the folder somewhere safe. E.g., the Documents folder.
4. Run Virtual Box.
5. Select *Add Machine* from the *Machine* submenu of the main menu.
6. Navigate to the folder containing the unzipped archive, select the `ThorV?.vbox` file, and *Open* it. Once the process completes, *ThorV?* will appear in the left panel of the Virtual Box window.

2.2.2 Running the Virtual Machine

1. Run Virtual Box. *ThorV?* should be visible in the left panel.
2. Double click on *ThorV?* in the left panel, or select it, and then click on *Start* in the top menu. A window like the one shown in Figure 2.1 will appear with text scrolling within. Once the message "`Starting Thor server`" appears, the server is running.



```
pf enabled
starting network
DHCPREQUEST on em0 to 255.255.255.255
DHCPACK from 10.0.2.2 (52:54:00:12:35:02)
bound to 10.0.2.15 -- renewal in 43200 seconds.
starting early daemons: syslogd pflogd.
starting RPC daemons:.
savecore: no core dump
checking quotas: done.
clearing /tmp
kern.securelevel: 0 -> 1
creating runtime link editor directory cache.
preserving editor files.
starting network daemons: smtpd sndiod.
***********************************************************************
*                                                                     *
*                         Running Thor                                *
*                                                                     *
***********************************************************************
starting local daemons: cron.
Tue Jan  5 08:58:26 AST 2016

OpenBSD/i386 (thor.cs.dal.ca) (ttyC0)

login: _
```

Figure 2.1: The *Thor* server is ready to go.

3. Run *Hammer* and change the *Server* setting in the *Arena toolbar* to `localhost` by using the drop-down list-box. *Hammer* will now use the local server instead of the remote one.

2.2.3 Shutting Down the Virtual Machine

To improve battery and system performance, shut down the *Thor* server when not in use.

1. Select *Quit* from the Virtual Box main menu, or use the (`command Q`) to quit.
2. When a dialog box comes up asking what to do, select *"Power off machine"*. This will shut down the server.
3. Select *Quit* from the Virtual Box main menu, or use the (`command Q`) to quit Virtual Box itself.

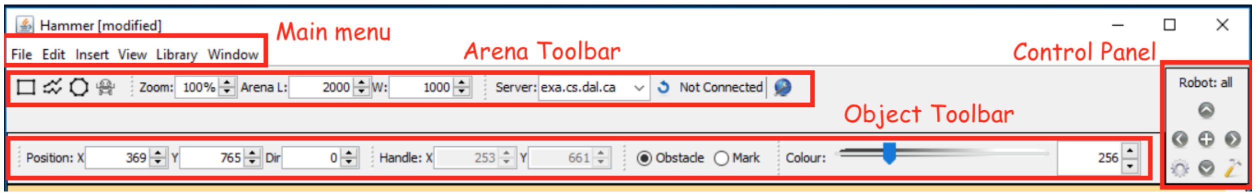
2.3 Installing and Running the Thor Server on a Unix Machine (optional)

Please see Chapter 8 for installation and usage instructions.

Chapter 3

Overview

The visualizer's user interface is divided into five parts: the arena, the application menu bar, the Arena toolbar, the Object toolbar, and the control panel for controlling the robot(s). The arena is the large yellow area where the majority of the activity takes place. The arena depicts the robot (or robots) and the simulated environment, which is initially a flat surface $1m \times 2m$ in size (approximately the size of a large wooden table). The user can create an environment for the robot(s) to negotiate, and then watch as the robots move through it. The application menu bar, provides functions for loading, storing, and editing the environment. The Arena toolbar allow the user to add artifacts to the arena, manipulate the arena, and connect to the *Thor* server. The Object toolbar is used to manipulate the artifacts in the arena, and the Control panel is used to interact with the robot(s).



Zoom: 100% Arena L: 2000 W: 1000 Server: exa.cs.dal.ca

Position: X 369 Y 765 Dir 0 Handle: X 253 Y 661 Colour: 256

Figure 3.1: The visualizer's control elements.

3.1 The ToolBars

The *Arena toolbar* is divided into three parts. The left section contains four buttons for adding artifacts to the arena, which include blocks (square button), markings on the arena surface (path button), arbitrary polygons that behave either as blocks or as surface markings (polygon button), and additional robots (robot button).

The middle section of the *Arena toolbar* allows the user to zoom in or zoom out on the arena and to change the arena's dimensions.

The right section of the *Arena toolbar* manages connections to the *Thor* server. The *Server* field specifies which server to connect to, the *Reset* button resets the connection, the *Status* field displays current connection status, and the *Network Preferences* button reveals advanced network preferences and information.

The *Object toolbar* is used to manipulate the position, orientation, and other properties of artifacts in the arena. Since different artifacts have different propertiesm this is a dynamic toolbar that changes, depending on the object selected. For example, in Figure 1.1, the robot is selected and the *Object toolbar* displays the properties of the robot, and the *Object toolbar* Figure 3.1 displays the properties of a Polygon artifact.

3.2 The Robot Control Panel

The *Control panel*, located to the right of the toolbars, contains controls for controlling and interacting with the robot. For example, in Figure 3.1, the Control panel has the five buttons on the Thymio. Pressing them causes the corresponding button event to occur in the simulated Thymio. There are two additional buttons. The *tap* (hammer) button in the bottom right of the panel simulates a tap on the Thymio, causing the *tap* event to fire on the simulated Thymio. The *Robot preferences* button in the bottom left of the panel reveals a configuration panel for the robot.

3.3 Creating and Manipulating the Environment

The simulated environment, depicted in the arena, consists of the arena surface and one or more *artifacts*, which are representations of physical objects, such as blocks, surface marks, and robots. Note, the terms ”arena and ”environment” can be used interchangeably.

Artifacts can be added to the arena by clicking on the buttons on the left side of the Arena toolbar (*Block*, *Path* (mark), *Polygon*, or *Robot*) and placing the artifact in the arena. The shape, size, orientation, position, and colour of the artifacts can then be changed by selecting the desired artifact and manipulating it via its handles or the *Object toolbar* located to the right of the arena. A *block* is a 3D artifact that can be bumped into and can be detected by the robot's horizontal sensors. A *mark* is a flat 2D artifact that lies on the arena surface and can be detected by the robot's ground sensors. A *robot* is also a 3D artifact and can detected by another robot's horizontal proximity sensors.

The artifacts in the arena are based on real physical objects used in the lab. For example, the default block size is that of 14×2 Duplo block. And, the default path width and colour is that of standard electrical tape. The default robot is a model of the Thymio II robot. For most purposes, the defaults should suffice. However, they can be changed by clicking on the artifact to select it, and them either manipulating the artifact's handles, or editing it via the *Object toolbar*. Artifacts in the arena can be modified and manipulated at any time. Any changes are automatically sent to the simulator, so a block can be moved in front of a moving robot, just like in real life.

Not all physical objects are Duplo blocks or electrical tape marks. In some instances, blocks or marks of different shape or colour are needed. The Polygon is a general purpose artifact that can either be a block or a mark. It is used to represent objects in the environment that do not correspond to Duplo blocks or electrical tape marks. Both the Block and Path artifacts can be converted to Polygon artifacts by clicking on the action in the *Edit* menu.

Once an environment is created, it can be saved and reloaded in a later session.

3.4 The Application Menu

The *File* menu provides the standard operations for loading, saving, and printing the environment, as well as quitting. All the operations use standard key-bindings. Environments can be saved and reloaded at a later time.

The *Edit* menu provides the standard operations for manipulating the environment, including: Copy, Delete, Paste, Undo, Redo, Group, Ungroup, and Select All. All the operations use standard key bindings, which are shown in the menus. For example, an artifact can be deleted by selecting it (clicking on it), and then pressing the `delete`, `backspace`, or (`command X`) key.[1] A useful operation, *Reset Arena*, resets the position of all robots in the arena to their original positions. This is particularly useful for debugging, in that it is helpful if the robot(s) start in the same configuration each time the program runs.

The *Insert* menu provides operations for inserting artifacts into the environment. The artifacts include: Blocks (`command B`), Paths (`command L`), Polygons (`command G`), and Robots (`command T`). The toolbar below the main menu contains buttons for each insert operation.

The *View* menu provides the operations for zooming in (`command +`) and zooming out (`command -`), with the standard key bindings. As well, the action to reset the connection to the *Thor* simulator. This is useful if the previous connection attempt failed because the server or the network were down.

The *Library* menu provides the operations for creating, loading, using, and adding to libraries. Artifacts, or groups of artifacts, can be stored in a library and then reused in the current or different environment. Please see Chapter 7 for a complete description.

The *Window* menu allows access to functionality located in separate windows. At present, the only additional functionality available is the *Log window*, which displays status messages and any information generated by the application as it is running. It is predominantly used for debugging purposes.

3.5 The Simulation

It is important to note that the actual simulation is done by the Thor simulator. The Hammer visualizer simply renders the simulation of the robot for the user. The movement of the robot is governed strictly by the speed of the wheel motors, i.e., basic physics.

[1] `command` is the `ctrl` key in Windows and Linux, and the `apple` key on a Mac.

The Thor server simulates only basic physics. It does not simulate drag, or friction, and it assumes that all artifacts have infinite weight and are immobile. Thus, if a robot bumps into a block or another robot, it stops, and must back up. If the robot makes contact with a block or another robot, or moves beyond the boundary of the arena, it stops and displays a sad face. Moving the robot in bounds and out of contact with other object will remove the sad face and allow the robot to proceed. By default, there is also no noise in the simulation, meaning that the robot(s) may behave a little differently than in the real world. E.g., Unlike in the real world, if both wheels of the simulated robot have the same target speed, it moves in a straight line. To add noise to the simulation, see Chapter 6.

The visualizer also displays the same LED lights that an actual robot has, which makes debugging easier. The robot can be tapped, using the *Hammer* button, and controlled through the five buttons (*Up*, *Right*, *Down*, *Left*, and *Center*) in the *Control panel*.

The simulation is updated at 20 to 60 frames per second. Thus, the visualizer may create a noticeable load on the local system.

Chapter 4

Connecting to the Thor Server

Hammer relies on the *Thor* server to perform the actual simulation of the robots in the constructed environment. Thus, *Hammer* must connect, via the Internet, to a *Thor* server in order to function.

4.1 Connecting and Reconnecting

Hammer attempts to connect to a *Thor* server after Aseba Studio connects to it. The connection is made automatically, without any actions needed by the user. User action is only needed if the connection fails or if the user wishes to switch servers. Once the connection is made, *Hammer* will continue to use the connection until Aseba Studio disconnects, or the connection between it and the server is broken.

If the connection between *Hammer* and *Thor* is broken, *Hammer* will automatically attempt to reconnect. The broken connection is usually a transient problem, which is resolved by reconnecting to the server. Occasionally, *Hammer* fails to connect to *Thor*.

If the connection fails, *Hammer* will cease efforts to connect after three attempts. Once the network problem is resolved (see Section 4.4) the user can attempt reconnection by clicking on the *Reconnect* button in the *Arena toolbar*. *Hammer* will then make three attempts to connect with the *Thor* server.

4.2 Changing Servers

By default, *Hammer* uses a remote *Thor* server (`exa.cs.dal.ca`). In many cases, it may be desirable to have *Hammer* use an alternate server. E.g., either a local server or a different remote server. Please see Chapter 2 for instructions on installing a local *Thor* server.

To change the *Thor* server, either select a different server from the drop-down list of the `Server` field in the *Arena toolbar*, or enter the new server's host name in the field. Once Aseba Studio connects to *Hammer*, *Hammer* will attempt to connect to the new server. If the connection is successful, the server becomes the new default.

4.3 Network Preferences and Information

The clicking on the *Network Preferences* button of the *Arena toolbar* reveals a preference panel where the user can change the default connection ports and view counts for the number of packets sent and received. Clicking on the button again will hide the preferences panel.

4.3.1 Aseba Studio Port

The *Aseba Studio Port* field contains the port on which *Hammer* listens for connections from Aseba Studio. By default, this port is 33333. Typically, this field will never need to be changed. But, if another application is using the same port, a different port may need to be used. Changing the field and clicking the *Reconnect* button will cause *Hammer* to listen on the new port. Be sure that the same port is specified when running Aseba Studio, so that it connects to the correct port.

4.3.2 Thor Server Port

The *Thor Server Port* field contains the port on which *Thor* listens for connections from *Hammer*. By default, this port is 34333. Typically, this field will never need to be changed. But, if the target *Thor* server is listening on another port, changing this field to the same port number is required. Remember to click the *Reconnect* button to enable the change. Note: If a different *Thor* server is selected, remember to change this field back to 34333.

4.3.3 Packet Counts

The last two fields in the *Advanced* section display the counts for sent and received packets. This is for informational purposes only and may be a useful indicator for the load that *Hammer* is imposing on the local network connection.

4.4 Debugging Network Issues

Occasionally, *Hammer* will fail to connect to the *Thor* server. This is typically manifested in the following symptoms:

1. Aseba Studio attempting to reconnect to *Hammer*.
2. Aseba Studio not responding while waiting for *Hammer* to connect to *Thor*.
3. Error messages appearing in the *Log window* in *Hammer*.
4. The *Status* field in the *Arena toolbar* indicating a failure to connect.

Connection failures are typically caused by one of the following issues:

Thor **server is not running.** If the target *Thor* server is remote, the system's administrator needs to be contacted. However, running a local *Thor* server (see Chapter 2) is an option.

Incorrect *Thor* server selected. Check the *Server* field in the *Arena toolbar* to ensure that *Hammer* is using the correct server. This is one of the most common issues.

Not connected to the Internet. If a remote *Thor* server is to be used, a working Internet connection is required. Fix the network connection before proceeding. Alternatively, running a local *Thor* server will avoid this problem (see Chapter 2).

Incorrect firewall settings. Occasionally, incorrect firewall settings on the local machine may cause connections to and from *Hammer* to be blocked. Check the firewall configuration and ensure that *Hammer* can receive connections on port 33333 and can create outbound connections on port 34333.

Incorrect ports settings (unlikely). If a port setting in the *Network Preferences* panel are incorrectly set, connections will not complete. See the previous section on how to change the settings.

Chapter 5

Artifacts and the Arena

Development and action takes place in the arena (environment), where the robot simulation occurs. The arena comprises a rectangular (yellow) surface on which the simulation takes place and one or more artifacts, which include the robots, blocks, surface marks, and groups of artifacts.

5.1 The Arena

The arena is the main yellow panel in *Hammer*. The default size of the arena is 1000×2000 millimeters, approximately the size of a large table. By default, the arena is displayed at full size, with a scale of one pixel per millimeter. However, this can be adjusted by changing the zoom factor.

Both the zoom factor and the arena dimensions are displayed in the *Arena toolbar*. The dimensions can be changed by changing the *width* and *length* fields. The zoom factor can be adjusted either via the *Arena toolbar*, the zoom in/out actions in the *View* menu, or hot-keys: (`command +`) to zoom in and (`command -`) to zoom out.

5.2 Artifacts

An Artifact is any object that can be placed on the surface of the arena. Artifacts include robots, blocks, paths, polygons, and groups of artifacts. Artifacts can be moved around the arena and can be sensed by robots.

An artifact is manipulated by selecting it (clicking on it) and dragging it or one of its handles with the pointer. An artifact can also be manipulated by directly modifying some its properties via the *Object toolbar*. For example, to change the direction of an artifact, a user can either select and drag the direction handle of the artifact or modify the *Dir.* field in the *Object toolbar*.

Artifacts share some properties but have specialized properties as well. For example, all artifacts have a position (x, y) and a direction (in degrees, where 0 is up). Some artifacts also have a colour, such as blocks, paths, and polygons, while other artifacts, such as robots and groups, do not.

The colour of a block, path, or polygon is represented by a single integer between 0 and 1024 on a grey scale, where 0 is black and 1024 is white. This is because the robot's sensors respond to the amount of light and do not distinguish colours. Please see Chapter 9 for a discussion of how the Thymio-II sensors are modeled. The colour of an artifact can be changed using the slider in the *Object toolbar*.

Artifacts can be created via the tool bar below the main menu or via the *Insert* menu. All artifacts, except robots, can be copied (`command C`), pasted (`command V`), deleted (`command X`), and grouped (`command alt-G`) using the standard *Edit* menu actions. Similarly, all artifacts, except robots, can be moved off the arena or can be only partially on the arena. Robot artifacts must always be on the arena.

5.2.1 The Block Artifact

Block artifacts are used to represent standard obstacles that can be sensed by the robot and must be avoided. The default Block artifact represents a forest green, 2×14 Duplo block ($32mm \times 223mm$).

A Block artifact is created by clicking on the *Block* button in the toolbar (`command B`) and dragging the block to its desired position. Block properties include position, direction, colour, length, and width. The latter two properties can be changed either via the *Object toolbar* or by selecting and moving one of the four corner handles.

When the colour of block is changed, only the outline of the block changes colour. The main fill remains green. This is to remind the user that the block is a 3D object so that only the outer surface can be sensed by a robot's horizontal proximity sensors.

As mentioned previously, the block is assumed to have infinite mass. That is, it cannot be moved during a collision with a robot. In other words, the block is assumed to be glued to the surface and hence unmovable.

5.2.2 The Path Artifact

Path artifacts are used to represent standard 2D marks on the surface of the arena that can be sensed by the robot. A Path artifact consists of one or more connected line segments of arbitrary length that are approximately 10mm wide and black in colour. I.e., line segments represent straight segments of electrical tape on the arena's surface.

A Path artifact is created by clicking on the *Path* button in the toolbar (`command L`) and dragging the starting end of the first segment to the desired position. Each following click places the other end of the current segment and the beginning of the next segment. A double click places the end of the last segment.

Path properties include position, direction, and colour. Each end-point of a segment has a handle that can be moved to change the shape of the path. The entire Path artifact is encased in a wire frame with a rotation handle, allowing the entire path to be rotated around the geographic center of the wire frame.

When the colour of path is changed, the entire path changes colour. This is to remind the user that the path is a 2D object, whose entirety can be sensed by a robot's ground proximity sensors. As mentioned previously, a robot cannot collide with a path because the path is a 2D object.

5.2.3 The Polygon Artifact

The Polygon artifact is a generalization of the Block and Path artifacts. In many instances, it may be useful to create obstacles or marks that are not shaped like blocks or paths. A Polygon artifact is a closed nonintersecting chain of three or more line segments, and can either be a 3D obstacle or a 2D surface mark. By default the Polygon artifact is a 3D obstacle and has the same forest green colour as a Block artifact.

A Polygon artifact is created by clicking on the *Polygon* button in the toolbar (`command G`) and dragging the first vertex to the desired position. Each following click places next vertex and a double click places the last vertex of the polygon. The vertices are automatically connected in counter-clockwise order and self-intersection is disallowed.

Polygon properties include position, direction, colour, and type. The polygon type is either *obstacle* (3D) or *mark* (2D). If the polygon is an obstacle, the boundary denotes the colour of the polygon, but the inner fill is forest green, just like the Block artifact. If the polygon is a mark, then it is rendered in its intended colour (black by default) just like a Path artifact.

Each vertex of a polygon has a handle that can be moved to change the shape of the polygon. The Polygon artifact is encased in a wire frame with a rotation handle, allowing the polygon to be rotated around the geographic center of the wire frame.

When the polygon is an obstacle, it behaves like a Block Artifact. I.e., it can be sensed by the robot's horizontal proximity sensors and is a physical obstacle for the robot. When the polygon is a mark, it behaves like a Path artifact. I.e., it is only sensed by the robot's ground proximity sensors.

5.2.4 The Group Artifact

The Group artifact comprises two or more artifacts that can be moved, rotated, copied, pasted, deleted, etc, in unison. The Group artifact is analogous to a group in Power Point, or various drawing applications. All kinds of artifacts may be part of a group, except robots. Artifacts can be mixed. For example, a Block, Path, and Polygon artifacts can be grouped together into one Group artifact.

A Group artifact is created by selecting multiple artifacts. Multiple artifacts can be selected by clicking on each artifact while holding down the `shift` key. Once all the desired artifacts are selected, use the *Group* action (`command alt-G`) in the *Edit* menu to create the Group artifact.

Group properties include position and direction. The Group artifact is encased in a wire frame with a rotation handle, allowing the group to be rotated around the geographic center of the wire frame. A group can be ungrouped by using the *Ungroup* action (`command shift-alt-G`) in the *Edit* menu.

Properties of individual artifacts within the group cannot be manipulated while the artifacts are in a group. To change an artifact within the group is it necessary to ungroup the group, modify the artifact in question, and then regroup the artifacts.

Chapter 6

The Thymio-II Simulated Robot

The purpose of *Thor* and *Hammer* is to simulate a robot in a standard environment. This chapter describes (i) how to manipulate, interact, and configure the robot in the simulation; (ii) how to create multi-robot simulations, which are much more interesting than a one-robot simulation; and (iii) how to use Aseba Studio in multi-robot mode.

6.1 Manipulating the Robot

Robots are manipulated in the same manner as other artifacts. They can be dragged and dropped in the arena and their direction can be changed by dragging the robot's handle.

6.1.1 Selecting and Moving the Robot

To select a robot, simply click on it. A wire frame should appear around the robot with single handle on the front. The robot's position, direction, and identifier will be displayed in the Object Panel on the right. To change the robot's position, drag the robot using the mouse pointer, or change the x and y coordinates in the *Object toolbar*. Note: the origin $(0, 0)$ is in the bottom left corner of the arena. Similarly, to change the direction of the robot, drag the handle or change the Direction field (in degrees) in the *Object toolbar*.

A robot can be dragged through or intersect a block. However, the robot is considered to be "in collision" with the block and will not move until the robot ceases to collide with the block. A robot cannot be dragged into or through another robot. Also, a robot cannot be dragged off the arena.

6.1.2 Resetting the Robot's Position

When debugging a program for some simulation, it will be necessary to run the robot through the same simulation multiple times. Typically, the robot will need to start from the same location each time. One way to do this is to drag the robot back to the start position each time. This can be error prone and tedious. A better way is to "reset" the robot's position using the *Reset Arena* action (`command R`) in the *Edit* menu. This changes the robot's position to the last saved position (see Chapter 7).

6.2 Interacting with the Robot

Interactions with the robot are limited to pressing the robot's buttons or tapping the robot. In the future, sound interactions may be possible. Interactions are performed via the Interactions Panel, located on the right. There are six buttons in the panel: five buttons corresponding to the buttons on the physical robot (*forward*, *backward*, *left*, *right*, *center*), and a *tap* (hammer) button, which generates a `tap` event on the simulated robot. A `tap` event also occurs when a robot collides with a block or another robot.

Note: The physical Thymio-II robot may occasionally miss button presses if the buttons are pressed too quickly. The simulated Thymio-II has a similar response.

6.3 Motor and Sensor Noise

By default, the motors and sensors of a simulated Thymio-II have no noise. However, a physical Thymio-II robot has some noise in both its sensors and motors. Noise refers to the variability and bias of the sensors and motors. For example, a motor's actual speed may vary randomly over time from its set speed (variability) and it may (on average) be a little faster or slower than the set speed (bias). Similarly, a sensor's response may vary (randomly) over time to a fixed input (variability) and the response may be (on average) higher or lower than the expected response for a given input. It is possible to configure the simulation of the robot to include noise in the motors and sensors.

To configure the robot's noise levels, click on the `Robot Preference` button in the bottom left of the *Control panel*. This will reveal a *Robot Preferences*, which contains the *Motor and Sensor Noise* settings.

6.3.1 Adding Bias to the Motors

Use the fields labeled *Left Motor Bias* and `Right Motor Bias` in the *Robot Preferences* panel to set the bias. The bias is specified as a percentage of the intended speed and ranges between -100% and 100%. However, the recommended range (with respect to reality) is in the neighbourhood of -10% to 10%. Note that the bias can be set independently for the left and right motors.

6.3.2 Adding Variability to the Motors

Use the fields labeled *Left Motor Var.* and *Right Motor Var.* in the *Robot Preferences* panel to set the bias. Variability is modeled using a Normal distribution with mean of the intended speed and a standard deviation that is a percentage of the intended speed, ranging between

0% and 100%. The recommended range is up to 10%. Note that the variability can be set independently for the left and right motors.

6.3.3 Adding Bias to the Ground Sensors

Use the field labeled *Ground Sensor Bias* in the *Robot Preferences* panel to set the bias. The bias is specified as an additive value to the response of the sensor, and ranges from -1000 and 1000. However, the response range of the sensor is 0 to 1000. The recommended range is between -50 to 50. Note: all ground sensors will exhibit the same bias.

6.3.4 Adding Variability to the Ground Sensors

Use the field labeled *Ground Sensor Var.* in the *Robot Preferences* panel to set the variability. Variability is modeled by an additive factor to the sensors' response function, which ranges from 0 to 1000. The additive factor is sampled from a Normal distribution with mean of 0 and a standard deviation of the specified size, up to 1000. A realistic setting should not exceed 5. Note: all ground sensors will exhibit the same variability.

6.3.5 Adding Bias to the Horizontal Sensors

Use the field labeled *Horizontal Sensor Bias* in the *Robot Preferences* panel to set the bias. The bias is specified as an additive value to the response of the sensor, and ranges from -5000 and 5000. However, the response range of the sensor is 0 to 5000. The recommended range is between -300 to 300. Note: all horizontal sensors will exhibit the same bias.

6.3.6 Adding Variability to the Horizontal Sensors

Use the field labeled *Horizontal Sensor Var.* in the *Robot Preferences* panel to set the variability. Variability is modeled by an additive factor to the sensors' response function, which ranges from 0 to 5000. The additive factor is sampled from a Normal distribution with mean of 0 and a standard deviation of the specified size, up to 5000. A realistic setting should not exceed 50. Note: all horizontal sensors will exhibit the same variability.

6.4 The Robot Identifier

Each robot has a numeric identifier, which is depicted in the center of each robot and in the *Object toolbar* and *Control panel* when the robot is selected. In an arena with only one robot, the robot identifier is not very useful. But, in an arena with multiple robots, the Robot Identifier is used to distinguish between the robots.

Robot identifiers must be unique in the arena, i.e., two robots in the same arena must have different robot identifiers. By default, *Hammer* assigns identifier 1 to the first robot, identifier 2 to the second robot, etc. It is also possible to change to robot's identifier by clicking on the *Change ID* button in the Object Panel, and selecting a new identifier. However, this should not be done without a good reason, as it may confuse Aseba Studio. In most cases the default Robot Identifier assigned by *Hammer* will suffice.

The Robot Identifier is accessible by the robot's program through a predefined hidden variable (`_id`). and can be used by the robot's program to make decisions. For example, a follow-the-leader program could use the identifier to decide which robot is the leader and which robots are followers. Alternatively, different programs can be loaded on the robots, one to lead and one to follow.

6.5 Multi-robot Simulations

In the real world, using multiple robots requires the availability of multiple robots. This may not be affordable or feasible in some cases. Fortunately, *Hammer* facilitates the multi-robot simulations without imposing a cost on the user.

6.5.1 Adding and Removing Robots in a Simulation

To add a robot to a simulation simply click on the "Robot" button in the toolbar (`command T`) and drag the robot to the desired location in the arena. *Hammer* will automatically assign the robot a unique identifier, and Aseba Studio will automatically add an additional tab where the program for the new robot is to be written.

To remove a robot from the simulation, select the robot and delete it like any other artifact, i.e., the `delete` key, (`command X`), or the *Cut* action from the *Edit* menu. Note that the corresponding tab in Aseba Studio will not be removed because Aseba has no way of distinguishing between a deleted robot and one that is simply slow in responding.

6.5.2 Manipulating, Interacting, and Configuring Robots

Manipulating and configuring robots in a multi-robot environment is accomplished in the same way as a single robot: Select the target robot by clicking on it, and then move it or configure it. Once the robot is selected, its Robot Identifier will appear in both the *Control panel* and the *Object toolbar*.

In many instances, it is useful to interact with all the robots in the environment at once. For example, in a race scenario it is useful to start all robots at the same time by pressing one of their buttons. To interact with all the robots at the same time ensure that none of the robots are selected. This can be accomplished by clicking on an empty spot in the arena. The *Control panel* will display a "`Controlling: All`" message. Consequently, when one of the buttons is pressed in the panel, the button event will be sent to all the robots in quick succession. I.e., the robots will not receive the events at exactly the same time, but within milliseconds of each other.

6.5.3 Using Aseba Studio

Lastly, a couple comments are in order on how Aseba Studio deals with multiple robots. For each robot Aseba Studio opens a tab where a program can be entered. Robots (can) have different programs, but share the same constants and global events. Unfortunately, there are a few design quirks.

First, the tabs all have the same name (`thymio-II`) and the only way to distinguish them is to look at the `_id` variable in the variable window. This makes programming multiple robots more challenging, especially if the programs are similar.

Second, programs for multiple robots are stored together in one program (`.aesl`) file. When the program is reloaded, Aseba Studio will complain if it does not detect the expected number of robots.

Third, programs are bound to a particular Robot Identifier, so if the Robot Identifiers in the program file do not match the Program Identifiers of the robots in *Hammer*, Aseba Studio will complain. In this situation, changing the robot identifiers in *Hammer* to match those in the program is the best solution.

Fourth, a single tab cannot be used to program all the robots. Even if all robots run the same program, it is necessary to copy the same program into each tab.

6.6 Advanced Options

The *Advanced* section of the *Robot Preferences* panel contains options governing the behaviour and rendering of the robot. At present, there is only one option in the section. This option allows the user to disable the rendering (drawing) of the buttons and LEDs on the robot. Not rendering the buttons and LEDs reduces the amount of time it takes to redraw the arena, which affects the performance of the *Hammer* application. If the application is running on an older or heavily loaded machine, or involves many robots, this option will improve the performance of the system. However, any use of LEDs by the program, will not be shown.

Chapter 7

Environments, Libraries, and Models

Hammer can save and load the environments that users create for their simulations. This is useful given the complexity of some environments and the amount of time it takes to create them. It is also a useful way to ensure that all parties participating in the same robotics simulation are subject to the same environment. I.e., instead of each party creating a similar environment, a single environment can be created and distributed to all parties.

Hammer also allows the user to create artifact libraries, which store artifacts or groups of artifacts that the user may wish to reuse in the future. This is useful for reducing the work in building new environments. For example, when creating an obstacle course, being able to reuse obstacles from a previously created obstacle course would save significant time.

At present, one robot model is available in *Hammer*, the Thymio-II. However, *Hammer* is designed to support multiple robot models. Once models for other common robots are developed, they can be dynamically loaded into *Hammer*.

7.1 Loading and Saving Environments

Environments can be saved and loaded via the *File* menu. Environment files have the extension `.hmr` and are simple text files that can be viewed with any text editor. The environment includes: the arena dimensions, the position, orientation, size, and shape of all the artifacts in the environment, and the position and orientation of the robots. This environment is sent to the *Thor* simulator when *Hammer* connects to it, and all changes to the environment are propagated to the server. The environment can be saved at any time.

7.2 Libraries

Libraries are collections of artifacts that can be used and reused in the environment. For example, if you create an obstacle course, with a variety of different obstacle artifacts. Each of these can be saved in a library and reused in future obstacle courses. Either a single artifact or a group of artifacts can be stored in a library and given a unique name for future identification. Libraries are stored in `.lbr` files, which are also simple text files. Libraries must be loaded or created before they are used to store or retrieve artifacts.

7.2.1 Creating or Loading Libraries

The Library menu provides operations for creating, loading, storing, and using a library. When loading or creating a library the user must specify the path and filename of the library using a standard file chooser dialog box.

Note: Libraries are not automatically opened when *Hammer* starts up. The user must explicitly load any libraries that he or she wishes to use.

7.2.2 Storing Artifacts in a Library

To store and artifact in a library:

1. Select the artifact or artifacts that are to be stored. To select multiple artifacts, hold down the `shift` key while selecting the artifacts.
2. Select the `Add to Library` operation in the Library menu.
3. Select the target library and a name of the item to be added.
4. Click OK.

7.2.3 Retrieving Artifacts from a Library

To retrieve an item from the library:

1. Be sure the library is loaded.
2. Select the `Select from Library` operation in the Library menu.
3. Select the library from which the item is to be retrieved.
4. Select the item from the library.
5. Click OK.
6. Drag and drop the item where you would like it in the arena.

Note, a library stores artifacts as a group. When an item is retrieved from the library, it is retrieved as a single group. To edit individual artifacts, the group first needs to be ungrouped (`command alt-shift-G`).

7.3 Robot Models

At present only one robot model is supported. Once multiple robot models are available, use the *Load Robot* action from the *File* menu to load the model (a `.jar` file). Once the model is loaded, a selection list appears beside the *Robot* button, allowing the user to select which robot they wish to create.

Note: A robot model must also be supported by the *Thor* server. If the *Thor* server being used does not support the loaded model, the robot will be ignored in the simulation.

Chapter 8

The Thor Server

The *Thor* server is a Unix based server that has been compiled and tested on Linux, Mac OS, and OpenBSD systems. It should compile and run on most BSD and Linux derivatives. It may even compile and run on Solaris.

8.1 Building the Thor Server

To install and run a local version of the server directly on a Unix machine:

1. Checkout the latest aseba-master source from GitHub.

   ```
   git clone --recursive https://github.com/aseba-community/aseba.git
   ```

   Although you should not need them, general instructions for compiling Aseba are here `https://github.com/aseba-community/aseba`

2. Contact the maintainer of the server (`abrodsky@cs.dal.ca`) to obtain the tarball.

3. In the `targets` subdirectory of the `aseba` source tree, create a `thor` subdirectory.

   ```
   cd aseba/targets
   mkdir thor
   cd thor
   ```

4. Unzip the tarball in the `thor` directory and run `make`.

   ```
   make
   ```

5. If things work as planned, a `thor` executable will be built.

8.2 Running the Thor Server

To quickly get things running:

1. To run the server that only accepts local connections use the command `./thor`
2. To run the server that accepts remote connections use the command `./thor -r`

The general invocation of the server is `./thor [options]` where the options are described below.

`-d`
: Turn on debug mode, which prevents the server from forking, making it easy to debug with GDB.

`-h`
: Print a help message describing all the options.

`-l <log file>`
: Log messages to specified log file instead of `thor.log`, which is the default log file. If this is the last option and there is no log file specified, logging is turned off.

`-p <secs>`
: Output performance statistics to the screen and the log every *secs* seconds. This is useful for finding performance bottle necks, but does slightly increase the load on the server to do extra system calls to get the current time of day.

`-q`
: Use quiet mode. Do not output anything to the console.

`-r`
: Allow remote connections. By default, the server only accepts localhost connections, because most users will run this locally for their use only. Allowing remote connections requires this switch.

`-v1`
: Accept version 1 protocol connections. This is a deprecated mode, which is here for transition from early versions of the protocol. Do not use this unless you really know what you are doing.

Chapter 9

Thymio-II Models

This chapter briefly describes the models used by the *Thor* simulator to model the Thymio-II robot. Most of the models are first order (or lower) approximations, with an emphasis on speed and efficiency, rather than physical accuracy. The output of the motors and the response of the sensors can be biased or have a nonzero variability as described in Section 6.3.

9.1 Motor Actuators

The motor actuators are modeled as a motor with zero torque on a robot of zero mass, with tires that have a very high friction coefficient. Similar to the physical robot, the motors have a speed range of $-500 \ldots 500$, where a speed of 500 is equal to 20cm/s. Unless noise is enabled, the target speed will be the actual speed of each motor.

Additionally, the motors are modeled as time-based stepper motors, with each step taking 0.01 seconds. That is, if there is no obstacle in the robot's path, the motor will complete a full step every 1/100th of a second. If there is an obstacle in the robot's path, no rotation occurs.

9.2 Ground Proximity Sensors

The ground proximity sensor is an infrared sensor that emits an infra-red light and measures the amount of infrared light reflected from the surface. The sensor produces three values:

Ambient light is the amount of infrared entering the sensor when it is not emitting an infrared light.

Reflective light is the amount of infrared entering the sensor when it is emitting an infrared light.

Difference light is the reflective amount minus the ambient amount of light.

In the simulation, the response to the ambient light is 0, as is typically the physical response. Consequently, the reflective and difference responses are identical. The reflective response ranges from 0 for a black surface mark, to 1023 for a white surface mark. The response is linear, meaning that a mark with a gray value of 500 (out of 1024) will yield a response of 500. The arena surface yields a response of 850.

9.3 Horizontal Proximity Sensors

The horizontal proximity sensor is an infrared sensor that emits an infra-red light and measures the amount of infrared light reflected from the objects in front of it. The sensor produces a single value, representing the amount of the infrared light that it is receiving: 0 for no infrared light, up to a maximum of 5000.

The amount of infrared light reflected by an object depends on a variety of factors, including distance from the sensor, shape, cross-section, colour, and material. Past analysis[1] of physical robots derived a response function of the form

$$F(x) = \frac{m \cdot (c - {x_0}^2)}{x^2 - 2x_0 x + c}$$

where x is the distance of the object from the sensor along its center line, x_0 is the position of the maximum response, m is the maximum response, and c is a third parameter. The ENKI[2] simulator uses a variation of this response function:

$$F_E(\vec{x}) = F(x_{center}) + F(x_{left}) + F(x_{right}) - 2 * F(x_{center}/\cos 15^\circ)$$

where F is the function above and $\vec{x}$ consists of distances measured by rays cast along the center line of the sensor and 15° left and right of the center line.

Both of these models have weaknesses. The former does not take into account that the sensors have an aperture of about 15° to the left and right of the center line. The latter model assumes that the object being sensed fully covers the aperture. For example, the response to an object that is intersected by two of the three rays would be near 0, as the last term in the F_E has a significant canceling effect.

The model used by *Thor* casts five rays, equally spaced throughout the 30° aperture. For each ray, it computes the distance (x) to the nearest object that it intersects, and computes $F(x)$ if x is less than the range of the sensor (10cm), or 0 if $x >$ 10cm. The response of the sensor is the average of the responses for each of the rays. Empirically, this yields a more accurate response particularly for simulations where the apperture is only partially obscured. Thus, only the three parameters for $F(x)$: x_0, c, and m need to be described.

The x_0 parameter in $F(x)$ is fixed at 0. This simplifies the response function, without sacrificing physical realism, since the experimental value for the sensor on physical Thymio robots was determined to be $0.03cm$ (a negligible amount). This leaves the remaining two parameters c and m.

[1]Nicolas Dinh, “Simulation du robot Thymio II: Rapport de Projet Semestriel (Automne 2013)”, École Polytechnique Fédérale de Lausanne (technical report), 2014, Page 14.

[2]ENKI: `https://github.com/enki-community/enki/blob/master/enki/interactions/IRSensor.h`, retrieved on June 19, 2017

The physical response of the sensor depends not only on its distance from the object but also, in part, on the color of the object. For example, a black object will reflect much less infrared light than a white object. *Thor* uses a lookup table, indexed by colour g (grey scale $0 \ldots 1023$), to determine what c and m parameters to use.

The parameter m is the maximum response of the sensor. This response occurs at $x = x_0$. If m is a function of colour g, by fixing $x = x_0$, we can experimentally determine $m(g)$. Our model uses the following formula for $m(g)$.

$$m(g) = \begin{cases} 3900, & g < 40 \\ 5055 - \frac{46200}{g}, & 40 \leq g < 841 \\ 5000, & g > 840 \end{cases}$$

The parameter c is the third parameter of $F(X)$, which fixes the other end of the curve at the end range of the sensor. At the 10cm mark the response function becomes discontinuous, dropping from around 900 to 0. The third parameter fixes the curve at the point right before the discontinuity. To determine c with respect to g, we fixed $x = 10cm$, and experimentally determined $c(g)$. To simply derivation, we define a function $R(g)$, which describes the response of $F(x)$ at some fixed distance x with respect to colour:

$$R(g) = \begin{cases} 945, & g < 40 \\ 4355 - \frac{130200}{g}, & 40 \leq g < 841 \\ 4200, & g > 840 \end{cases}$$

which we can then use to solve for $c(g)$:

$$c(g) = \frac{625R(g)}{5000 - R(g)}$$

Lastly, the maximum distance of the sensor is also affected by the colour of the object—lighter the object the greater the maximum distance at which the sensor will sense it. Again, this function, $d_{max}(g)$, is determined experimentally

$$d_{max}(g) = \begin{cases} 30mm, & g < 40 \\ 100 - \frac{70}{209}(\frac{210}{\delta(g)} - 1), & 40 \leq g < 841 \\ 100mm, & g > 840 \end{cases}$$

where

$$\delta(g) = e^{\frac{\log(210)}{800}(g-40)}$$

9.4 Tap Sensor

The tap sensor on a physical robot requires some minimum acceleration to register as a tap by the robot's accelerometer. The simulated tap responds to any acceleration. I.e., any collision will trigger a tap.