Donnie Robot User Documentation

Release 0.1

Alexandre Amory

Apr 22, 2019

Contents

CHAPTER 1

About Donnie

./source/getting_started/images/donnie-robot.png

./source/getting_started/images/donnie-sim.png

Warning: This document is for users only. If you want to build, configure, or add features to your Donnie, please refer to the Donnie Assistive Robot: Developer Manual

Donnie is an assistive technology project, whose objective is to use robotics to facilitate programming teaching to visually impaired students. It is divided in two main parts:

- The construction and fine-tuning of the titular mobile robot, Donnie;
- The project's software stack, including an intuitive parser/interpreter and a robot simulation environment;

The project is in its second version, developed in the Laboratório de Sistemas Autônomos (LSA) of the Pontific Catholic University of Rio Grande do Sul (PUCRS), Brazil. Before going into the tutorials, we are assuming the programming environment is already configured. If it is not the case, please refer to the 'Donnie's Developer Manual'.

1.1 About Donnie

Robotics has been used to teach young students the basics of programming. However, most of the programming environments for kids are high visual, based on grab and drag blocks. Blind students or students with some visual disability cannot use these teaching resources.

The Donnie project proposes an inclusive robotic programming environment which all students (with or without visual disabilities) can use.

Donnie comes with two usage options: with the simulated and with the physical robots. It is recommend to start with simulation since it does not require building the robot. Moreover, the physical robot is functional, but still under test.

1.1.1 Features

- Robot programming environment for young students with or without visual impairment;
- Assistive programming language called GoDonnie. GoDonnie is TTS and screen reader friendly;
- Integration with a Arduino-based robot with Player robotic middleware;
- Extension of Stage simulator to generate sound clues while the robot is moving;
- Software developed for the simulated robot is compatible with the real Donnie robot;

The simulation is recommended if you want to known about Donnie but don't have the required resources to build your own Donnie robot.

1.2 Donnie Programming Environment

1.2.1 Introduction

GoDonnie is a programming language that commands a robot called Donnie. This robot works in its own environment. To open this environment it's necessary to take a few steps. Open a Linux terminal and type `donnie_player` and press `ENTER`. After that, in order to start the module of GoDonnie's commands type `GoDonnie -t`. These expressions can be typed in capital letter or lower case. Now, in the line of command, the user types the commands he wants. It's possible to type one command per line or several commands in the same line. After each line, the user should press `ENTER`. For the robot to execute the commands, it's necessary to press `ESC`. This way, the user can type a command and press `ENTER` and `ESC`, or he can type one command per line, pressing `ENTER` to change lines, and only pressing `ESC` when he wants to run the commands. The GoDonnie language doesn't difference capital letters to lower cases, and the commands can be typed with or without accentuation. The robot always starts at the point 0,0 facing north. This position is located on the left bottom. The commands in this manual are all in Portuguese.

1.2.2 GoDonnie Programming Language

GoDonnie User manual

Section 1: Leaving the programming terminal

Command: `QUIT EXIT`

Subject-matter: None

Explanation: It closes the programming environment. It can only be used in the terminal.

Example:

```
QUIT
```

Section 2: Declaration of variables

Variable is an object that holds value. This variable can only receive integer values

Command: `VAR x`

Subject-matter: `x` is the variable that is going to be created. This variable receives integer values.

Explanation: There are 5 forms to create a variable:

1. Create a variable;
2. Create a variable and assign an initial value;
3. Create a variable that receives an expression;
4. Create a variable inside the command of repeat `FOR`;
5. Create a variable that will receive a value from another command, such as: `COLOR` and `POS`.

Variables hold only the last received values. They store only integer values. This way, if there is a comma result, only the integer part will be stored in the variable.

There are rules for the name of the variables:

- There are no difference between capital letter and lower case;
- It can't contain special characters. Exemple: *, @, #, +;
- It can't initiate with a number. Exemple: VAR 52abc is wrong.

Example:

1. To create a variable without initial value, it's possible to do:

```
VAR A
```

Creates a variable called A.

```
A = 2
```

When a variable is created it's possible to assign directly a value. The variable called `A` will store the value 2.

2. To create a variable with initial value, it's possible to do:

```
VAR B = 5
```

Creates a variable called `B` and stores the value 5.

3. To create a variable that receives an expression, it's possible to do:

```
VAR C = A + B
```

Creates a variable called `C` that receives the value of the variable `A` added to the value of the variable `B`. The result of variable `C` is 7.

```
C = 1
```

Changes the value of the variable `C` and stores the value 1, losing its previous value.

4. To create a variable inside the command `FOR` (this command will be seen in section 10 of the manual), it's possible to do:

```
FOR VAR d = 0; d > 5; d = d +1 DO
FW 1
END FOR
```

The robot will take 5 steps forward.

5. To create a variable that receives value from another command, it's possible to do:

```
VAR d = DISTANCE F
VAR c = COLOR GREEN
VAR px = POS x
```

The variable `d` is going to store the front distance from the robot to the object. The variable `c` is going to store the number of green objects. And the variable `px` is going to store the current position of the robot in the x axis. (The commands `distance`, `color` and `pos` will be seen in section 10 of the manual).

```
G = 5
```

It will return an error because the variable `G` was not created.

Section 3: Audio Commands

Commands for manipulation and return of audio

a)

Command: `SPEAK x`

Subject-matter: `x` is a variable that must have been created previously.

Explanation: It speaks the content of the variable. This sound is issued by the robot or the virtual environment, it depends on which one is active.

Example:

```
VAR x = 5
SPEAK x
```

Will be spoken: 5

b)

Command: `SPEAK "x"`

Subject-matter: `x` is a word or a phrase, that needs to be between quotation marks

Explanation: It speaks the word or phrase between quotation marks. This sound is issued by the robot or the virtual environment, it depends on which one is active.

Example:

```
SPEAK "hello"
```

Will be spoken: hello

c)

Command: `SOUND on` `SOUND off`

Subject-matter: It turns the sound on or off

Explanation: These commands turn on and off the audio from the robot or the virtual environment.

Example:

```
SOUND ON
SOUND OFF
```

Section 4: Operators

They are operators that provide support for logical and mathematical expressions

Command: Operators

Subject-matter:

Mathematical:

```
+ addition
- subtraction
* multiplication
/ division
```

Comparators:

```
<> different
== equals
< is less than
> is more than
<= is less or equals than
>= is more or equals than
```

Assignment:

```
= assignment
```

Explanation: Operators are used to compare values or expressions.

Example: *To add*

```
Var a = 2
```

Creating the variable `a` and storing the value 2

```
Var b = 1
```

Creating the variable `b` and storing the value 1

```
Var add
```

Creating the variable `add`

```
add = a + b
```

Storing in `add` the sum from variables `a` and `b`

```
Speak add
```

Will be spoken: 3

To divide

```
Var c = 2
```

Creating the variable `c` and storing the value 2

```
Var d = 2
```

Creating the variable `d` and storing the value 2

```
Var division
```

Creating the variable `division`

```
division = c / d
```

Storing in `division` the division from variables `c` and `d`

```
Speak division
```

Will be spoken: 1

Section 5: Movement Commands

They are commands that move the robot in the environment

a)

Command: `FW n` `FORWARD n`

Subject-matter: `n` is the number of steps. This command accepts only integer and positive numbers, or variables that store integer numbers, or mathematical expressions that result in integer numbers.

Explanation: It walks `n` steps forward

Example:

```
FW 5
```

The robot will walk 5 steps forward

```
VAR A = 10
FW A
```

The robot will walk 10 steps forward

```
VAR A = 10
VAR B = 20
FW A + B
```

The robot will walk 30 steps forward

If the robot hits into something before it walks the number of steps, you will be informed: "`I walked only X steps forward. I found an obstacle.`"

```
FW -5
```

When a negative number is typed as a command, the user will be informed that the robot walked 0 steps.

b)

Command: `BW n BACKWARD n`

Subject-matter: `n` is the number of steps. This command accepts only integer and positive numbers, or variables that store integer numbers, or mathematical expressions that result in integer numbers.

Explanation: It walks `n` steps backward.

Example:

```
BW 5
```

The robot will walk 5 steps backward

```
VAR A = 10
BW A
```

The robot will walk 10 steps backward

```
VAR A = 10
VAR B = 20
BW A + B
```

The robot will walk 30 steps backward

If the robot hits into something before it walks the number of steps, you will be informed: "`I walked only X steps backward. I found an obstacle.`"

```
BW -5
```

When a negative number is typed as a command, the user will be informed that the robot walked 0 steps.

Section 6: Rotation Commands

The robot rotates without movement

a)

Command: `TR n RIGHT TURN n`

Subject-matter: `n` is the number of degrees. This command accepts only integer and positive numbers, or variables that store integer numbers, or mathematical expressions that result in integer numbers.

Explanation: Turns `n` degrees to the right. The robot does no displacement.

Example:

```
TR 90
```

The robot will turn 90 degrees to the right.

```
VAR A = 45
TR A
```

The robot will turn 45 degrees to the right.

```
VAR A = 80
VAR B = 10
TR A + B
```

The robot will turn 90 degrees to the right.

```
TR - 90
```

The robot will turn 90 degrees to the left.

b)

Command: `TL n LEFT TURN n`

Subject-matter: `n` is the number of degrees. This command accepts only integer and positive numbers, or variables that store integer numbers, or mathematical expressions that result in integer numbers.

Explanation: Turns `n` degrees to the left. The robot does no displacement.

Example:

```
TL 90
```

The robot will turn 90 degrees to the left.

```
VAR A = 45
TL A
```

The robot will turn 45 degrees to the left.

```
VAR A = 80
VAR B = 10
TL A + B
```

The robot will turn 90 degrees to the left.

```
TL - 90
```

The robot will turn 90 degrees to the right.

Section 7: Commands of Visualization of the Environment

These are commands to get informations about the environment in which the robot is inserted. It's not possible to store in variables the return from these commands

a)

Command: `SCAN`

Subject-matter: None

Explanation: It returns the identification of the object, the approximate angle and the approximate distance between the robot and the identified object. The tracking for the identification of objects occurs from 90 degrees to the left to 90 degrees to the right from the front.

Example: Let's assume that the robot is in the position 2,3, facing north, and there's a green obstacle in the position 0,5 and other red obstacle in the position 6,3.

```
SCAN
```

It will be spoken:

`To the 40 degrees in the left:  1 object of color green at 2 steps. To the 90 degrees in the right:  1 object of color red at 4 steps.`

In case two objects are in the same degree, it will inform: `To the 30 degrees in the left:  2 objects of color green, red at 17 steps.`

b)

Command: `STATE`

Subject-matter: None

Explanation: It returns the position in the X, Y axis and the angle from the robot. It also informs the last rotation or movement command that was typed before the command ESTADO.

Example:

```
FW 3 STATE
```

Let's say the robot was at 0,0. The robot will walk 3 steps forward and it will inform: `"I walked 3 steps forward, command 1 was PF 3, walked 3, didn't crash, position [3,0,0]."` The number 3 corresponds to the X axis, the first 0 corresponds to the Y axis and the last 0 corresponds to the angle of the robot.

In case the robot crashes into something and only completes 2 steps successfully, the command will return: `"I walked 3 steps forward, command 1 was PF 3, walked 2, crashed, position [2,0,0]."`

If there weren't previous commands, it will return: "`No command executed, position [0,0, 0]`".

Section 8: Commands of Position and Perception of the Environment

These are commands to get informations about the environment in which the robot is inserted. It is possible to store in variables the return from these commands

a)

Command: `DISTANCE d`

Subject-matter: `d` is the direction of the robot sensor (`f` - front; `fr` - frontal right; `fl` - frontal right; `b` - back; `bl` - back left; `br` - back right).

Explanation: It returns the quantity of steps from the robot sensor to an obstacle, depending on the chosen direction.

There are three ways to use the DISTANCE command:

1. If the user wants to have a feedback, he should use the command `SPEAK` with the command `DISTANCE`.
2. If the user wants only to store in a variable.
3. If the user wants to use it directly in another command, for example: `IF` (section 9), `FOR` (section 10), `REPEAT` (section 10) or `WHILE` (section 10).

- DISTANCE F returns the number of steps from the robot to an object detected by the front sensor of the robot.
- DISTANCE FR returns the number of steps from the robot to an object detected by the frontal right sensor of the robot.
- DISTANCE FL returns the number of steps from the robot to an object detected by the frontal left sensor of the robot.
- DISTANCE B returns the number of steps from the robot to an object detected by the back sensor of the robot.
- DISTANCE BR returns the number of steps from the robot to an object detected by the right rear sensor of the robot.
- DISTANCE BL returns the number of steps from the robot to an object detected by the left rear sensor of the robot.

If there aren't obstacles, it returns the quantity of step that the sensor can identify, that usually is 60 steps.

Example:

```
DISTANCE F
DISTANCE FR
DISTANCE FL
DISTANCE B
DISTANCE BL
DISTANCE BR
```

Assuming that the robot is in the position 0,0, facing north and there are obstacles in the following positions, the result will be:

Obstacle in 0,3:

```
SPEAK DISTANCE F
```

Answer: 3 Steps

You can previously create a variable, and after use it to store what the command DISTANCE will return:

```
VAR d = DISTANCE T
```

It stores in the variable `d` the back distance from the robot to the obstacle that is directly behind it. Assuming that the robot is in the position 0,3 facing north and there is an obstacle in 0,0. The stored value in `d` will be 3.

```
IF DISTANCE F>3 THEN
FW 1
ELSE
SPEAK "It's impossible to move forward"
END IF
```

In the example above, if the front distance from the robot is more than 3, the robot will move 1 step forward. If the distance is equal or less than 3, it will return: `"It's impossible to move forward"`

```
WHILE DISTANCE F>3
DO
FW 1
END WHILE
```

In the example above, while the front distance from the robot to the object is more than 3, it will move 1 step forward.

b)

Command: `POS k` `POSITION k`

Subject-matter: `k` is an axis of the Cartesian plane (X or Y) or angle (A).

Explanation: It returns the current position of the robot in the X axis or Y axis or the current angle of the robot.

There are three ways to use the POS k command:

1. If the user wants to have feedback, he should also use the command `SPEAK` altogether with the command `POS x`, `POS y` or `POS a`;
2. If the user wants only to store it in a variable;
3. If the user wants to use it directly in another command, for example: `IF` (section 9), `FOR` (section 10), `REPEAT` (section 10) or `WHILE` (section 10).

Example: If the user wants to have a feedback, he can do as it follows:

Assuming the robot is in the position 0,0 facing north:

```
SPEAK POS x
```

It will be spoken 0

```
SPEAK POS y
```

It will be spoken 0

```
SPEAK POS a
```

It will be spoken 0

If the user only wants to store the position value:

```
VAR z = POS x
```

The variable `z` has stored the value of the position of the robot in the x axis

```
VAR b = POS y
```

The variable `b` has stored the value of the position of the robot in the y axis

```
VAR i = POS a
```

The variable `i` contains the angle of the robot

If the user wants to use it inside other commands:

```
IF POS b > 0 THEN
FW 5
ELSE
BW 5
END IF
```

c)

Command: `COLOR c`

Subject-matter: `c` is the color you want (blue, red, green)

Explanation: Verifies how many objects of a certain color the robot can identify at an angle of 180 degrees ahead of it.

There are three ways to use the COR c command:

1. If the user wants to have a feedback, he should use the command `SPEAK` with the command `COLOR`.
2. If the user wants only to store in a variable.
3. If the user wants to use it directly in another command, for example: `IF` (section 9), `FOR` (section 10), `REPEAT` (section 10) or `WHILE` (section 10).

Example:

If the user wants to have a feedback, he can do as it follows:

Assuming there is one green object and two blue objects:

```
SPEAK COLOR blue
```

It will be spoken 2

```
SPEAK COLOR green
```

It will be spoken 1

If the user only wants to store the color value:

```
VAR A = COLOR BLUE
```

The variable A stores the number of blue objects.

```
VAR V = COLOR GREEN
```

The variable V stores the number of green objects.

If the user wants to use it inside other commands:

```
IF COLOR BLUE > 0 THEN
SPEAK "Number of blue objects"
SPEAK COLOR BLUE
ELSE
SPEAK "Couldn't find blue objects"
END IF

IF COLOR GREEN > 0 THEN
SPEAK "Number of green objects"
SPEAK COLOR GREEN
ELSE
SPEAK "Couldn't find green objects"
END IF
```

Section 9: Condition Commands

These are conditional commands that allow the program to choose what will be executed, according to the stipulated condition

a)

Command:

IF *expression logical operator expression*
THEN *commands*
ELSE *commands*
END IF

Subject-matter: Expression = variable or expression

Explanation: Test if a condition is true and, if it is,executes the first line of commands, else executes the ELSE line of commands.

Example: Assuming that, if the variable is less than 4 the robot has to walk 5 steps forward, else it has to turn 45 degrees to the left:

```
VAR a = 0
IF a<4
THEN FW 5
ELSE TL 45
END IF
```

b)

Command:

`IF` *expression logical operator expression*
`THEN` *commands*
`END IF`

Subject-matter: Expression = variable or expression

Explanation: Test if a condition is true and, if it is, executes the first line of commands.

Example:

```
VAR a = 0
IF a<4
THEN FW 5
END IF
```

If the variable `a` has a value less than 4 the robot will walk 5 steps forward

Section 10: Repeat commands

These are commands that allow one or more instructions to be executed a certain number of times

a)

Command:

`FOR` *initialization; expression logical operator expression; increment or decrement*
`DO` *commands*
`END FOR`

Subject-matter:

Initialization: variable = an integer value

Variable or expression logical operator variable or expression: variable or expression - logical operator - variable or expression.

Increment: variable + constant or variable + variable
Decrement: variable - constant or variable - variable

Explanation: Repeats the commands a certain number of times

Example: The example makes the robot to walk towards an obstacle that's in front of him and after each step he says "hello".

```
VAR obstacle = DISTANCE F
FOR VAR x=1; x<=obstacle; x=x+1
DO
FW 1
SPEAK "hello"
END FOR
```

The variable `x` will start with 1 as value, the robot will walk one step forward and will say "hello" while the value is less or equal the distance to the obstacle.

b)

Command:

`REPEAT n TIMES` commands
`END REPEAT`

Subject-matter: `n` is the number of times that the commands will be repeated

Explanation: Repeats the commands `n` times

Example:

```
REPEAT 4 TIMES
TR 90
FW 2
END REPEAT
```

Assuming the robot will start at the position 0,0 , the commands TR 90 and FW 2 will be repeated 4 times. The robot's trajectory will look like a square.

c)

Command:

`WHILE` *expression logical operator expression*
`DO` *commands*
`END WHILE`

Subject-matter: Variable or expression logical operator variable or expression: variable or expression - logical operator - variable or expression.

Explanation: Repeats the commands while expression - logical operator - expression is true.

Example: This example makes the robot walk towards an obstacle in front of him and after each step speak "I'm coming".

```
WHILE DISTANCE > 3
DO
FW 1
SPEAK "I'm coming"
END WHILE
```

While the distance ftom the front of the robot in relation to the object is more than 3, the robot will walk one step forward and will speak "I'm coming".

Section 11: Declaration of procedures

Procedure is a smaller program(subprogram) that allows decompose and solve a more complex problem in a simpler. It can be called in other parts of the program

a)

Command:

`PROCEDURE` *name: variable1, variable2, variable3, ...*
`DO` *commands*
`END PROCEDURE`

Subject-matter: Name is the name of the subprogram and variable1, variable2, variable3 are its arguments

Explanation: It creates a subprogram. This command only works via file.

Example: The robot needs to walk simulating a rectangle. This rectangle can have different sizes, according to the activity. Therefore the command PROCEDURE can be used to create a procedure called RECTANGLE that receives two variables, one for the height and the other for the base. Thus this procedure can be used to make rectangles with different sizes.

```
PROCEDURE RECTANGLE: base, height
DO
FW base TR 90
FW height TR 90
FW base TR 90
FW height TR 90
END PROCEDURE
```

Or

```
PROCEDURE RECTANGLE: base, height
DO
REPEAT 2 TIMES
FW base TR 90
FW height TR 90
END REPEAT
END PROCEDURE
```

Calling the subprogram

```
RECTANGLE [5,3]
RECTANGLE [8,4]
RECTANGLE [9,5]
```

Section 12: Various Commands

a)

Command:

`WAIT t`

Subject-matter: `t` is the time in seconds

Explanation: Waits `t` seconds to run the next commands

Example: If the robot needs to walk forward 2 steps, wait 3 seconds and then walk 4 more steps

```
FW 2
WAIT 3
FW 4
```

b)

Command: –

Subject-matter: None

Explanation: After this symbol – everything that is in the line that has – won't be executed. These are reminders about the code.

Example:

```
-- This is a comment
```

1.2.3 GoDonnie Interpreter

modos de operacao, exemplos de uso

1.3 Donnie Robot Simulator

nao falar sobre como montar um novo cenario. para isso, aponte para o manual do desenvolvedor. mencionar brevemente que o simulador eh baseado no Stage.

1.3.1 Donnie Scenarios

falar sobre os cenarios prontos.

um dos cenarios deve incluir muitiplos robos.

2.1 - Important File Types

In Player/Stage there are 3 kinds of file that you need to understand to get going with Player/Stage:

- a .world file
- a .cfg (configuration) file
- a .inc (include) file

The .world file tells Player/Stage what things are available to put in the world. In this file you describe your robot, any items which populate the world and the layout of the world. The .inc file follows the same syntax and format of a .world file but it can be *included*. So if there is an object in your world that you might want to use in other worlds, such as a model of a robot, putting the robot description in a .inc file just makes it easier to copy over, it also means that if you ever want to change your robot description then you only need to do it in one place and your multiple simulations are changed too.

The .cfg file is what Player reads to get all the information about the robot that you are going to use.This file tells Player which drivers it needs to use in order to interact with the robot, if you're using a real robot these drivers are built in to Player (or you can download or write your own drivers, but I'm not going to talk about how to do this here.)

Alternatively, if you want to make a simulation, the driver is always Stage (this is how Player uses Stage in the same way it uses a robot: it thinks that it is a hardware driver and communicates with it as such). The .cfg file tells Player how to talk to the driver, and how to interpret any data from the driver so that it can be presented to your code. Items described in the .world file should be described in the .cfg file if you want your code to be able to interact with that item (such as a robot), if you don't need your code to interact with the item then this isn't necessary. The .cfg file does all this specification using interfaces and drivers, which will be discussed in the following section.

2.2 - Interfaces, Drivers and Devices

- Drivers are pieces of code that talk directly to hardware. These are built in to Player so it is not important to know how to write these as you begin to learn Player/Stage. The drivers are specific to a piece of hardware so, say, a laser driver will be different to a camera driver, and also different to a driver for a different brand of laser. This is the same as the way that drivers for graphics cards differ for each make and model of card. Drivers produce and read information which conforms to an **interface**. The driver design is described in Chapter 11.
- Interfaces are a set way for a driver to send and receive information from Player. Like drivers, interfaces are also built in to Player and there is a big list of them in the Player manual. They specify the syntax and semantics of how drivers and Player interact. The interface design is described in Chapter 10.
- A device is a driver that is bound to an interface so that Player can talk to it directly. This means that if you are working on a real robot that you can interact with a real device (laser, gripper, camera etc) on the real robot, in a simulated robot you can interact with their simulations.

The official documentation actually describes these three things quite well with an example. (Actually, the official documentation still refers to the depreciated laser interface, but I've updated all the references in this manual to use the new ranger interface.)

> Consider the ranger interface. This interface defines a format in which a planar range-sensor can return range readings (basically a list of ranges, with some meta-data). The ranger interface is just that: an interface. You can't do anything with it.
>
> Now consider the sicklms200 driver. This driver controls a SICK LMS200, which is particular planar range sensor that is popular in mobile robot applications. The sicklms200 driver knows how to communicate with the SICK LMS200 over a serial line and retrieve range data from it. But you don't want to access the range data in some SICK-specific format. So the driver also knows how to translate the retrieved data to make it conform to the format defined by the ranger interface.
>
> The sicklms200 driver can be bound to the ranger interface ... to create a device, which might have the following address:
>
> `localhost:6665:ranger:0`
>
> The fields in this address correspond to the entries in the `player_devaddr_t` structure: host, robot, interface, and index. The host and robot fields (localhost and 6665) indicate where the device is located. The interface field indicates which interface the device supports, and thus how it can be used. Because you might have more than one ranger, the index field allows you to pick among the devices that support the given interface and are located on the given host:robot Other lasers on the same host:robot would be assigned different indexes.

The last paragraph there gets a bit technical, but don't worry. Player talks to parts of the robot using ports (the default port is 6665), if you're using Stage then Player and Stage communicate through these ports (even if they're running on the same computer). All this line does is tell Player which port to listen to and what kind of data to expect. In the example it's laser data which is being transmitted on port 6665 of the computer that Player is running on (localhost). You could just as easily connect to another computer by using its IP address instead of ''localhost". The specifics of writing a device address in this way will be described in Chapter 4

Fig. 1: img

3.1 - Building an Empty World

When we tell Player to build a world we only give it the .cfg file as an input. This .cfg file needs to tell us where to find our .world file, which is where all the items in the simulation are described. To explain how to build a Stage world containing nothing but walls we will use an example.

To start building an empty world we need a .cfg file. First create a document called `empty.cfg` (i.e. open in your favorite text editor - `gedit` is a good starter program if you don't have a favorite) and copy the following code into it:

```
driver
(
   name "stage"
   plugin "stageplugin"

   provides ["simulation:0" ]

   # load the named file into the simulator
   worldfile "empty.world"
)
```

The configuration file syntax is described in Chapter 4, but basically what is happening here is that your configuration file is telling Player that there is a driver called `stage` in the `stageplugin` library, and this will give Player data which conforms to the `simulation` interface. To build the simulation Player needs to look in the worldfile called `empty.world` which is stored in the same folder as this .cfg. If it was stored elsewhere you would have to include a filepath, for example `./worlds/empty.world`. Lines that begin with the hash symbol (#) are comments. When you build a simulation, any simulation, in Stage the above chunk of code should always be the first thing the configuration file says. Obviously the name of the worldfile should be changed depending on what you called it though.

Now a basic configuration file has been written, it is time to tell Player/Stage what to put into this simulation. This is done in the .world file.

3.1.1 - Models

A worldfile is basically just a list of models that describes all the stuff in the simulation. This includes the basic environment, robots and other objects. The basic type of model is called "model", and you define a model using the following syntax:

```
define model_name model
(
   # parameters
)
```

This tells Player/Stage that you are `defining` a `model` which you have called `model_name`, and all the stuff in the round brackets are parameters of the model. To begin to understand Player/Stage model parameters, let's look at the `map.inc` file that comes with Stage, this contains the `floorplan` model, which is used to describe the basic environment of the simulation (i.e. walls the robots can bump into):

```
define floorplan model
(
  # sombre, sensible, artistic
  color "gray30"

  # most maps will need a bounding box
  boundary 1
```

(continues on next page)

(continued from previous page)

```
  gui_nose 0
  gui_grid 0
  gui_move 0
  gui_outline 0
  gripper_return 0
  fiducial_return 0
  ranger_return 1
)
```

We can see from the first line that they are defining a `model` called `floorplan`.

- `color`: Tells Player/Stage what colour to render this model, in this case it is going to be a shade of grey.
- `boundary`: Whether or not there is a bounding box around the model. This is an example of a binary parameter, which means the if the number next to it is 0 then it is false, if it is 1 or over then it's true. So here we DO have a bounding box around our "map" model so the robot can't wander out of our map.
- `gui_nose`: this tells Player/Stage that it should indicate which way the model is facing. Figure 3.2 shows the difference between a map with a nose and one without.
- `gui_grid`: this will superimpose a grid over the model. Figure 3.3 shows a map with a grid.
- `gui_move`: this indicates whether it should be possible to drag and drop the model. Here it is 0, so you cannot move the map model once Player/Stage has been run. In 1.4 - Try It Out when the Player/Stage example `simple.cfg` was run it was possible to drag and drop the robot because its `gui_move` variable was set to 1.
- `gui_outline`: indicates whether or not the model should be outlined. This makes no difference to a map, but it can be useful when making models of items within the world.
- `fiducial_return`: any parameter of the form some_sensor_return describes how that kind of sensor should react to the model. "Fiducial" is a kind of robot sensor which will be described later in *Section 3.2 - Fiducial*. Setting `fiducial_return` to 0 means that the map cannot be detected by a fiducial sensor.
- `ranger_return`: Setting `ranger_return` to a negative number indicates that a model cannot be seen by ranger sensors. Setting `ranger_return` to a number between 0 and 1 (inclusive) (Note: this means that `ranger_return 0` **will allow** a ranger sensor to see the object — the *range* will get set, it'll just set the *intensity* of that return to zero.) See Section 5.3.2 - Interaction with Proxies — Ranger for more details. controls the intensity of the return seen by a ranger sensor.
- `gripper_return`: Like `fiducial_return`, `gripper_return` tells Player/Stage that your model can be detected by the relevant sensor, i.e. it can be gripped by a gripper. Here `gripper_return` is set to 0 so the map cannot be gripped by a gripper.

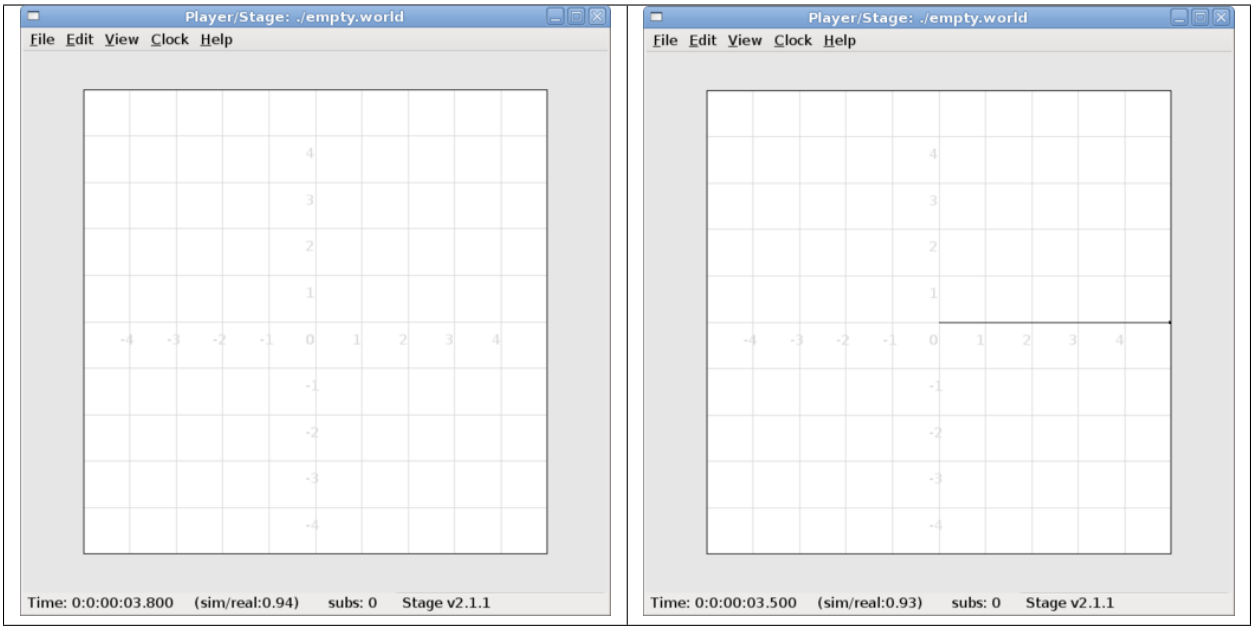

Figure 3.2: The first picture shows an empty map without a nose. The second picture shows the same map with a nose to indicate orientation, this is the horizontal line from the centre of the map to the right, it shows that the map is actually facing to the right.

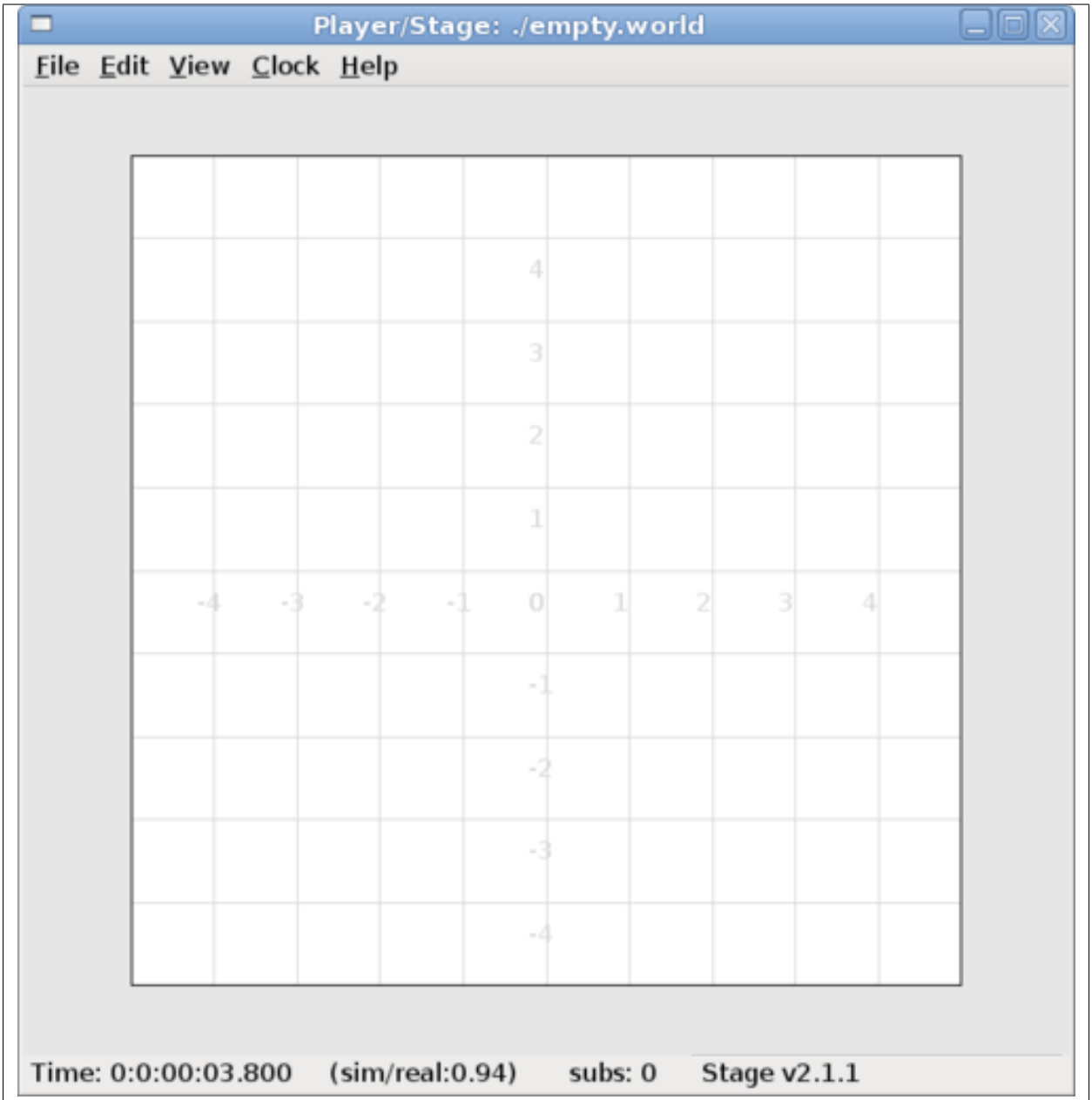

Figure 3.3: An empty map with gui_grid enabled. With gui_grid disabled this would just be an empty white square.

To make use of the `map.inc` file we put the following code into our world file:

```
include "map.inc"
```

This inserts the `map.inc` file into our world file where the include line is. This assumes that your worldfile and `map.inc` file are in the same folder, if they are not then you'll need to include the filepath in the quotes. Once this is done we can modify our definition of the map model to be used in the simulation. For example:

```
floorplan
(
   bitmap "bitmaps/helloworld.png"
```

(continues on next page)

(continued from previous page)

```
    size [12 5 1]
)
```

What this means is that we are using the model "floorplan", and making some extra definitions; both "bitmap" and "size" are parameters of a Player/Stage model. Here we are telling Player/Stage that we defined a bunch of parameters for a type of model called "floorplan" (contained in map.inc) and now we're using this "floorplan" model definition and adding a few extra parameters.

- `bitmap`: this is the filepath to a bitmap, which can be type bmp, jpeg, gif or png. Black areas in the bitmap tell the model what shape to be, non-black areas are not rendered, this is illustrated in Figure 3.4. In the map.inc file we told the map that its "color" would be grey. This parameter does not affect how the bitmaps are read, Player/Stage will always look for black in the bitmap, the `color` parameter just alters what colour the map is rendered in the simulation.
- `size`: This is the size *in metres* of the simulation. All sizes you give in the world file are in metres, and they represent the actual size of things. If you have 3m x 4m robot testing arena that is 2m high and you want to simulate it then the `size` is [3 4 2]. The first number is the size in the *x* dimension, the second is the *y* dimension and the third is the *z* dimension.

Figure 3.4: The first image is our "helloworld.png" bitmap, the second image is what Player/Stage interprets that bitmap as. The coloured areas are walls, the robot can move everywhere else.

A full list of model parameters and their descriptions can be found in the official Stage manual Most of the useful parameters have already been described here, however there are a few other types of model which are relevant to building simulations of robots, these will be described later in *Section 3.2 - Building a Robot*.

3.1.2 - Describing the Player/Stage Window

The worldfile also can be used to describe the simulation window that Player/Stage creates. Player/Stage will automatically make a window for the simulation if you don't put any window details in the worldfile, however, it is often useful to put this information in anyway. This prevents a large simulation from being too big for the window, or to increase or decrease the size of the simulation.

Like a model, a window is an inbuilt, high-level entity with lots of parameters. Unlike models though, there can be only one window in a simulation and only a few of its parameters are really needed. The simulation window is described with the following syntax:

```
window
(
   # parameters...
)
```

The two most important parameters for the window are `size` and `scale`.

- `size`: This is the size the simulation window will be *in pixels*. You need to define both the width and height of the window using the following syntax: `size [width height]`.
- `scale`: This is how many metres of the simulated environment each pixel shows. The bigger this number is, the smaller the simulation becomes. The optimum value for the scale is window_size/floorplan_size and it should be rounded downwards so the simulation is a little smaller than the window it's in, some degree of trial and error is needed to get this right.

A full list of window parameters can be found in the Stage manual under "WorldGUI"

3.1.3 - Making a Basic Worldfile

We have already discussed the basics of worldfile building: models and the window. There are just a few more parameters to describe which don't belong in either a model or a window description, these are optional though, and the defaults are pretty sensible.

- `interval_sim`: This is how many simulated milliseconds there are between each update of the simulation window, the default is 100 milliseconds.
- `interval_real`: This is how many real milliseconds there are between each update of the simulation window. Balancing this parameter and the `interval_sim` parameter controls the speed of the simulation. Again, the default value is 100 milliseconds, both these interval parameter defaults are fairly sensible, so it's not always necessary to redefine them.

The Stage manual contains a list of the high-level worldfile parameters

Finally, we are able to write a worldfile!

```
include "map.inc"

# configure the GUI window
window
(
   size [700.000 700.000]
   scale 41
)

# load an environment bitmap
floorplan
(
   bitmap "bitmaps/cave.png"
   size [15 15 0.5]
)
```

If we save the above code as empty.world (correcting any filepaths if necessary) we can run its corresponding empty.cfg file (see *Section 3.1 - Empty World*) to get the simulation shown in Figure 3.5.

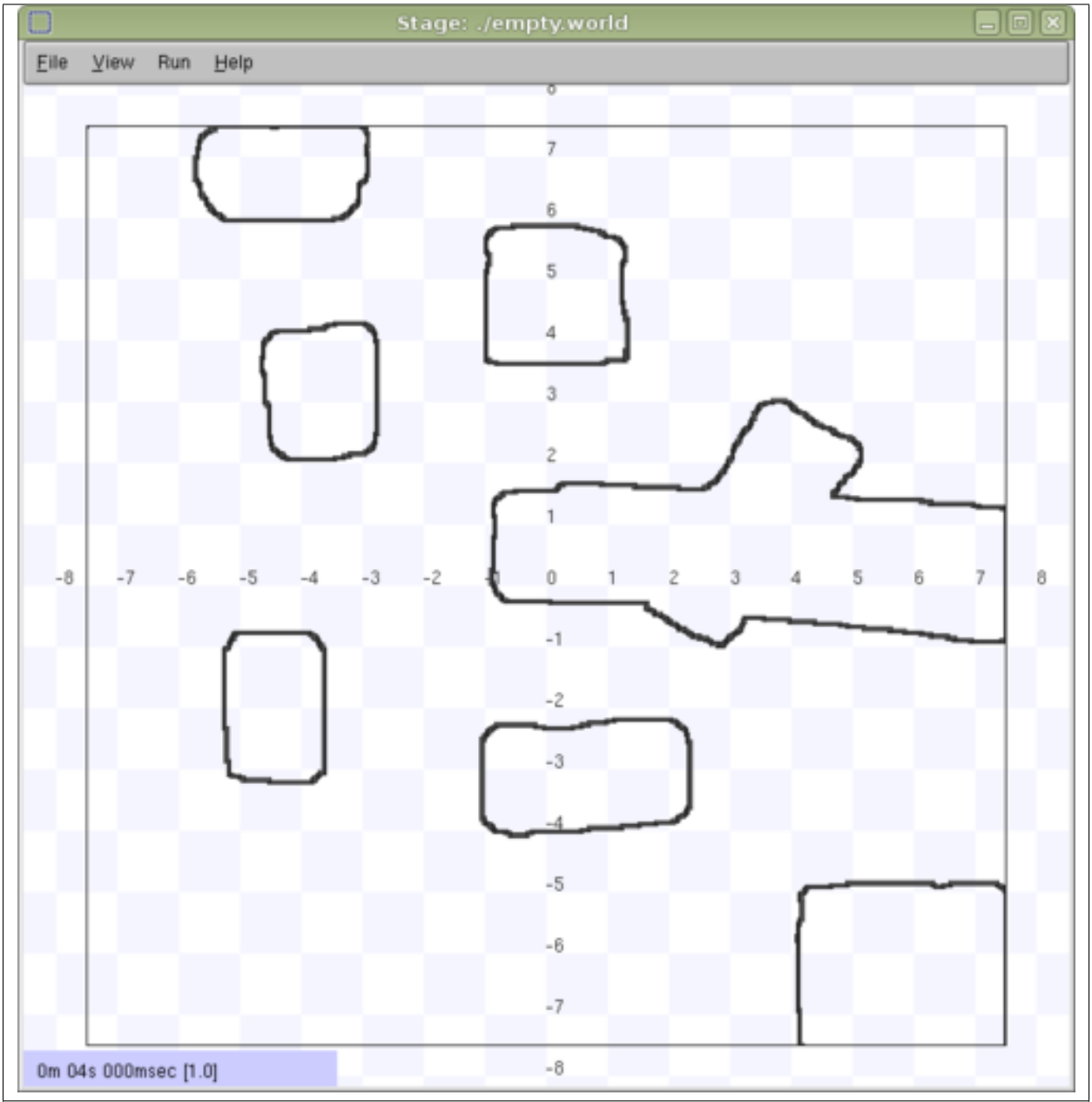

Figure 3.5: Our Empty World

3.2 - Building a Robot

In Player/Stage a robot is just a slightly advanced kind of model, all the parameters described in *Models* can still be applied.

3.2.1 - Sensors and Devices

There are six built-in kinds of model that help with building a robot, they are used to define the sensors and actuators that the robot has. These are associated with a set of model parameters which define by which sensors the model can be detected (these are the `_returns` mentioned earlier). Each of these built in models acts as an *interface* (see

Section 2.2 - Interfaces, Drivers, and Devices) between the simulation and Player. If your robot has one of these kinds of sensor on it, then you need to use the relevant model to describe the sensor, otherwise Stage and Player won't be able to pass the data between each other. It is possible to write your own interfaces, but the stuff already included in Player/Stage should be sufficient for most people's needs. A full list of interfaces that Player supports can be found in the Player manual although only the following are supported by the current distribution of Stage (version 4.1.X). Unless otherwise stated, these models use the Player interface that shares its name:

3.2.1.1 - camera

The camera model adds a camera to the robot model and allows your code to interact with the simulated camera. The camera parameters are as follows:

- `resolution [x y]`: the resolution, in pixels, of the camera's image.
- `range [min max]`: the minimum and maximum range that the camera can detect
- `fov [x y]`: the field of view of the camera *in DEGREES*.
- `pantilt [pan tilt]`: angle, in degrees, where the camera is looking. Pan is the left-right positioning. So for instance pantilt [20 10] points the camera 20 degrees left and 10 degrees down.

3.2.1.2 - blobfinder

The blobfinder simulates colour detection software that can be run on the image from the robot's camera. It is not necessary to include a model of the camera in your description of the robot if you want to use a blobfinder, the blobfinder will work on its own.

In previous versions of Stage, there was a `blob_return` parameter to determine if a blobfinder could detect an object. In Stage 4.1.1, this does not seem to be the case. However, you can simply set an object to be a color not listed in the `colors[]` list to make it invisible to blobfinders.

The parameters for the blobfinder are described in the Stage manual, but the most useful ones are here:

- `colors_count <int>`: the number of different colours the blobfinder can detect
- `colors [ ]`: the names of the colours it can detect. This is given to the blobfinder definition in the form `["black" "blue" "cyan"]`. These colour names are from the built in X11 colour database rgb.txt. This is built in to Linux – the file `rgb.txt` can normally be found at /usr/share/X11/rgb.txt assuming it's properly installed, or see Wikipedia for details.
- `image [x y]`: the size of the image from the camera, in pixels.
- `range <float>`: The maximum range that the camera can detect, in metres.
- `fov <float>`: field of view of the blobfinder *in DEGREES*. Unlike the camera `fov`, the blobfinder `fov` respects the `unit_angle` call as described in http://playerstage.sourceforge.net/wiki/Writing_configuration_files#Units. By default, the blobfinder `fov` is in DEGREES.

3.2.1.3 - fiducial

A fiducial is a fixed point in an image, so the fiducial finder simulates image processing software that locates fixed points in an image. The fiducialfinder is able to locate objects in the simulation whose `fiducial_return` parameter is set to true. Stage also allows you to specify different types of fiducial using the `fiducial_key` parameter of a model. This means that you can make the robots able to tell the difference between different fiducials by what key they transmit. The fiducial finder and the concept of `fiducial_keys` is properly explained in the Stage manual. The fiducial sensors parameters are:

- `range_min`: The minimum range at which a fiducial can be detected, in metres.
- `range_max`: The maximum range at which a fiducial can be detected, in metres.
- `range_max_id`: The maximum range at which a fiducial's key can be accurately identified. If a fiducial is closer that `range_max` but further away than `range_max_id` then it detects that there is a fiducial but can't identify it.
- `fov`: The field of view of the fiducial finder *in DEGREES*.

3.2.1.4 - ranger sensor

The ranger sensor simulates any kind of obstacle detection device (e.g. sonars, lasers, or infrared sensors). These can locate models whose `ranger_return` is non-negative. Using a ranger model you can define any number of ranger sensors and apply them all to a single device. The parameters for the `sensor` model and their inputs are described in the Stage manual, but basically:

- `size [x y]`: how big the sensors are.
- `range [min max]`: defines the minimum and maxium distances that can be sensed.
- `fov deg`: defines the field of view of the sensors in DEGREES
- `samples`: this is only defined for a laser - it specifies ranger readings the sensor takes. The laser model behaves like a large number of rangers sensors all with the same x and y coordinates relative to the robot's centre, each of these rangers has a slightly different yaw. The rangers are spaced so that there are samples number of rangers distributed evenly to give the laser's field of view. So if the field of view is 180 and there are 180 samples the rangers are 1 apart.

3.2.1.5 - ranger device

A ranger device is comprised of ranger sensors. A laser is a special case of ranger sensor which allows only one sensor, and has a very large field of view. For a ranger device, you just provide a list of sensors which comprise this device, typically resetting the pose for each. How to write the `[x y yaw]` data is explained in *Yaw Angles*.

```
sensor_name (pose [x1 y1 z1 yaw1])
sensor_name (pose [x2 y2 z2 yaw2])
```

3.2.1.6 - gripper

The gripper model is a simulation of the gripper you get on a Pioneer robot. The Pioneer grippers looks like a big block on the front of the robot with two big sliders that close around an object. If you put a gripper on your robot model it means that your robot is able to pick up objects and move them around within the simulation. The online Stage manual says that grippers are deprecated in Stage 3.X.X, however this is not actually the case and grippers are very useful if you want your robot to be able to manipulate and move items. The parameters you can use to customise the gripper model are:

- `size [x y z]`: The x and y dimensions of the gripper.
- `pose [x y z yaw]`: Where the gripper is placed on the robot, relative to the robot's geometric centre. The pose parameter is decribed properly in *Section 3.2.1 - Robot Sensors and Devices*.

3.2.1.7 - position

The position model simulates the robot's odometry, this is when the robot keeps track of where it is by recording how many times its wheels spin and the angle it turns. This robot model is the most important of all because it allows the robot model to be embodied in the world, meaning it can collide with anything which has its `obstacle_return` parameter set to true. The position model uses the `position2d` interface, which is essential for Player because it tells Player where the robot actually is in the world. The most useful parameters of the position model are:

- `drive`: Tells the odometry how the robot is driven. This is usually "diff" which means the robot is controlled by changing the speeds of the left and right wheels independently. Other possible values are "car" which means the robot uses a velocity and a steering angle, or "omni" which means it can control how it moves along the x and y axes of the simulation.
- `localization`: tells the model how it should record the odometry "odom" if the robot calculates it as it moves along or "gps" for the robot to have perfect knowledge about where it is in the simulation.
- `odom_error [x y angle]`: The amount of error that the robot will make in the odometry recordings.

3.2.2 - An Example Robot

To demonstrate how to build a model of a robot in Player/Stage we will build our own example. First we will describe the physical properties of the robot, such as size and shape. Then we will add sensors onto it so that it can interact with its environment.

3.2.2.1 - The Robot's Body

Let's say we want to model a rubbish collecting robot called "Bigbob". The first thing we need to do is describe its basic shape, to do this you need to know your robot's dimensions in metres. Figure 3.6 shows the basic shape of Bigbob drawn onto some cartesian coordinates, the coordinates of the corners of the robot have been recorded. We can then build this model using the `block` model parameter. In this example we're using blocks with the position model type but we could equally use it with other model types.

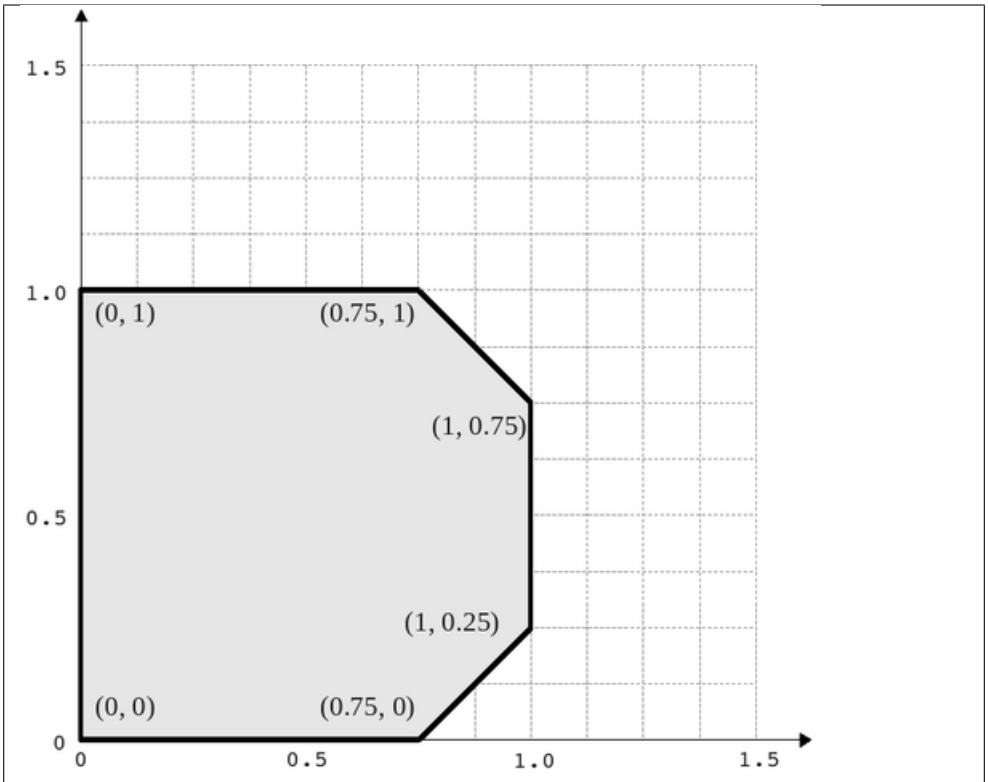

Figure 3.6: The basic shape we want to make Bigbob, the units on the axes are in metres.

```
define bigbob position
(
      block
      (
            points 6
            point[0] [0.75 0]
            point[1] [1 0.25]
            point[2] [1 0.75]
            point[3] [0.75 1]
            point[4] [0 1]
            point[5] [0 0]
            z [0 1]
      )
)
bigbob
(
          name "bob1"
          pose [ 0 0 0 0]
          color "gray"
)
```

In the first line of this code we state that we are defining a `position` model called `bigbob`. Next `block` declares that this `position` model contains a block.

The following lines go on to describe the shape of the block; `points 6` says that the block has 6 corners and `point[number] [x y]` gives the coordinates of each corner of the polygon in turn. Finally, the `z [height_from height_to]` states how tall the robot should be, the first parameter being a lower coordinate in the z plane, and the second parameter being the upper coordinate in the z plane. In this example we are saying that

the block describing Bigbob's body is on the ground (i.e. its lower z coordinate is at 0) and it is 1 metre tall. If I wanted it to be from 50cm off the ground to 1m then I could use `z [0.5 1]`.

TRY IT OUT (Position Model)

In this example, you can see the basic shape in an empty environment.

3.2.2.2 - Adding Teeth

Now in the same way as we built the body we can add on some teeth for Bigbob to collect rubbish between. Figure 3.7 shows Bigbob with teeth plotted onto a cartesian grid:

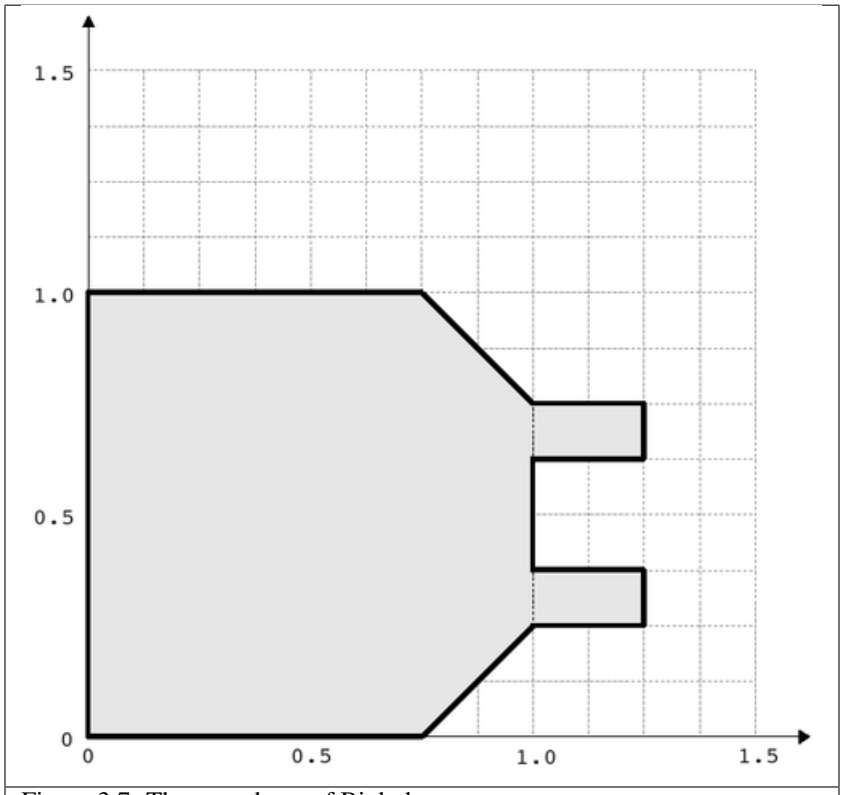

Figure 3.7: The new shape of Bigbob.

```
define bigbob position
(
      size [1.25 1 1]

      # the shape of Bigbob

      block
      (
            points 6
            point[5] [0 0]
            point[4] [0 1]
            point[3] [0.75 1]
            point[2] [1 0.75]
```

(continues on next page)

(continued from previous page)

```
            point[1] [1 0.25]
            point[0] [0.75 0]
            z [0 1]
      )

      block
      (
            points 4
            point[3] [1 0.75]
            point[2] [1.25 0.75]
            point[1] [1.25 0.625]
            point[0] [1 0.625]
            z [0 0.5]
      )

      block
      (
            points 4
            point[3] [1 0.375]
            point[2] [1.25 0.375]
            point[1] [1.25 0.25]
            point[0] [1 0.25]
            z [0 0.5]
      )
)
```

To declare the size of the robot you use the `size [x y z]` parameter, this will cause the polygon described to be scaled to fit into a box which is `x` by `y` in size and `z` metres tall. The default size is 0.4 x 0.4 x 1 m, so because the addition of rubbish-collecting teeth made Bigbob longer, the size parameter was needed to stop Player/Stage from making the robot smaller than it should be. In this way we could have specified the polygon coordinates to be 4 times the distance apart and then declared its size to be `1.25 x 1 x 1` metres, and we would have got a robot the size we wanted. For a robot as large as Bigbob this is not really important, but it could be useful when building models of very small robots. It should be noted that it doesn't actually matter where in the cartesian coordinate system you place the polygon, instead of starting at `(0, 0)` it could just as easily have started at `(-1000, 12345)`. With the `block` parameter we just describe the *shape* of the robot, not its size or location in the map.

TRY IT OUT (BigBob with Teeth)

This example shows the more accurate rendering of Big Bob.

3.2.2.3 - Yaw Angles

You may have noticed that in Figures 3.6 and 3.7 Bigbob is facing to the right of the grid. When you place any item in a Player/Stage simulation they are, by default, facing to the right hand side of the simulation. Figure 3.3 shows that the grids use a typical Cartesian coordinate system, and so if you want to alter the direction an object in the simulation is pointing (its "yaw") any angles you give use the x-axis as a reference, just like vectors in a Cartesian coordinate system (see Figure 3.8) and so the default yaw is *0* degrees. This is also why in *Section 3.1 - Empty World* the `gui_nose` shows the map is facing to the right. Figure 3.9 shows a few examples of robots with different yaws.

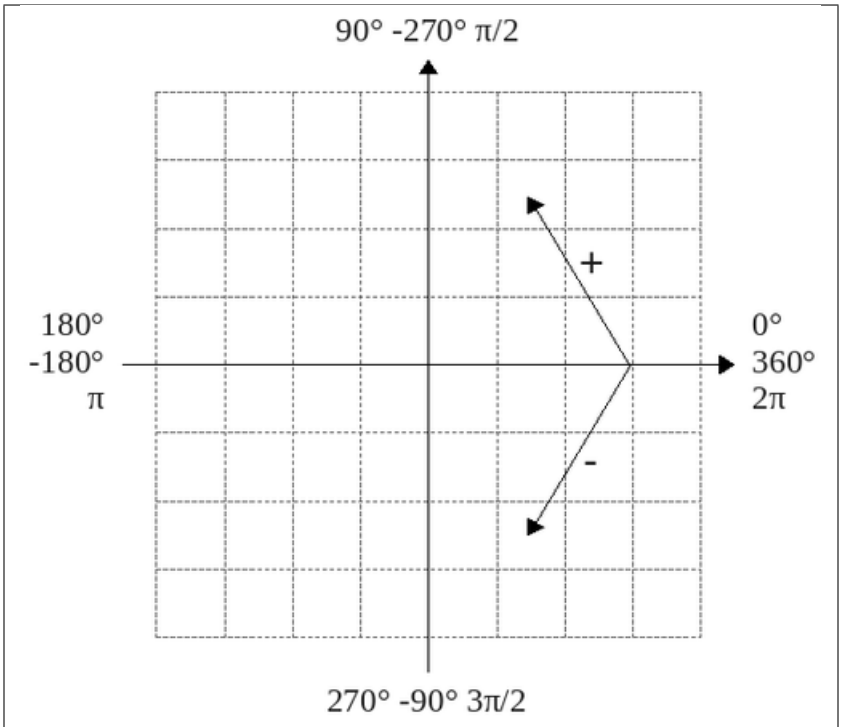

Figure 3.8: A cartesian grid showing how angles are described.

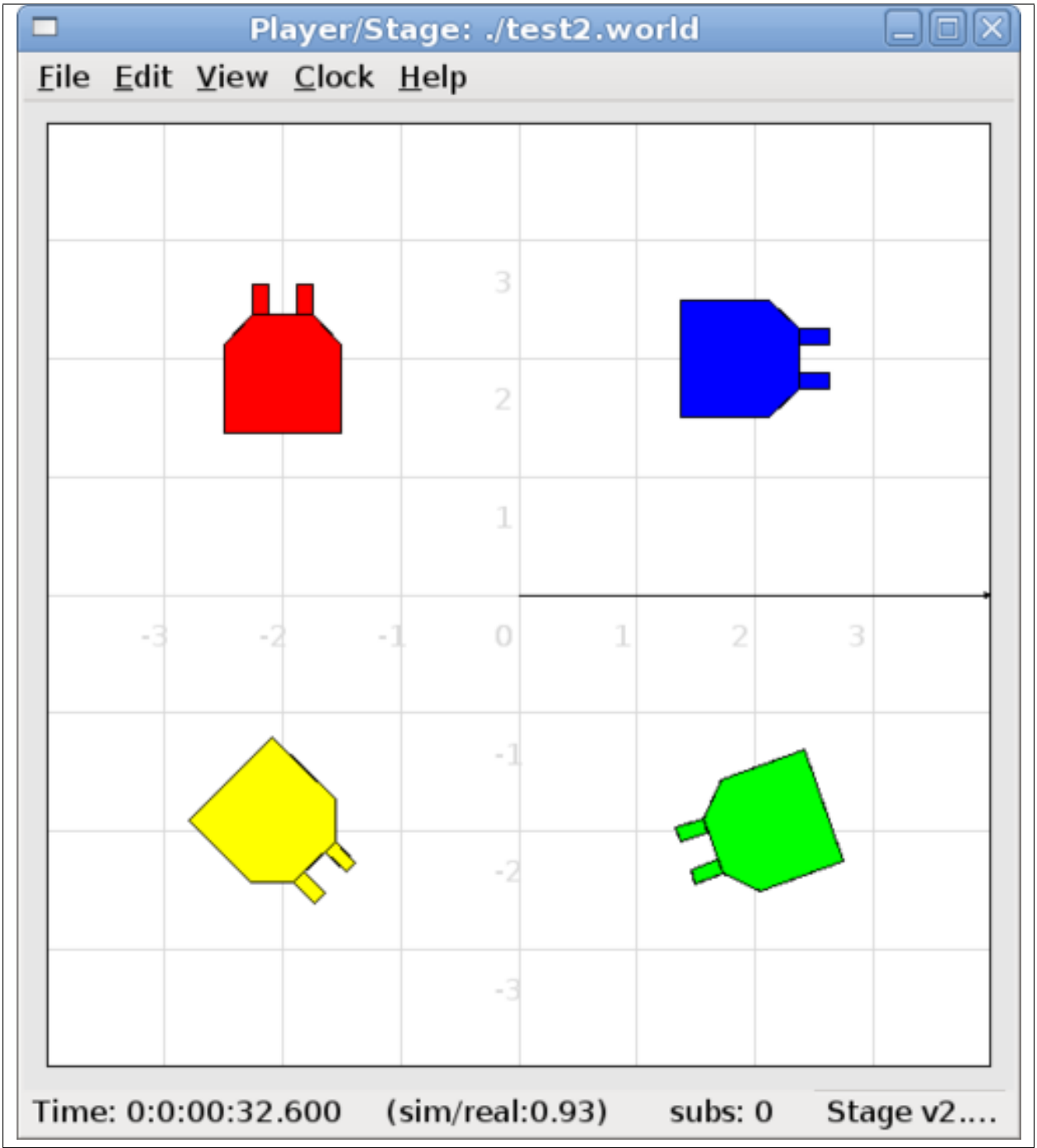

Figure 3.9: Starting from the top right robot and working anti-clockwise, the yaws of these robots are 0, 90, -45 and 200.

By default, Player/Stage assumes the robot's centre of rotation is at its geometric centre based on what values are given to the robot's `size` parameter. Bigbob's `size` is `1.25 x 1 x 1` so Player/Stage will place its centre at `(0.625, 0.5, 0.5)`, which means that Bigbob's wheels would be closer to its teeth. Instead let's say that Bigbob's centre of rotation is in the middle of its main body (shown in Figure 3.6 which puts the centre of rotation at `(0.5, 0.5, 0.5)`. To change this in robot model you use the `origin [x-offset y-offset z-offset]` command:

```
define bigbob position
(
      # actual size
      size [1.25 1 1]
      # centre of rotation offset
      origin [0.125 0 0]

      # the shape of Bigbob
      block
            ...
            ...
            ...
)
```

TRY IT OUT (Different Origin)

Click on the robot, and it should hilight. You can drag bigbob around with the left (primay) mouse button. Click and hold down the right (secondary) mouse button, and move the mouse to rotate bigbob about the centre of the body, not the centre of the entire block.

3.2.2.4 - Drive

Finally we will specify the `drive` of Bigbob, this is a parameter of the `position` model and has been described earlier.

```
define bigbob position
(
      # actual size
      size [1.25 1 1]
      # centre of rotation offset
      origin [0.125 0 0]

      # the shape of Bigbob
      block
            ...
            ...
            ...

      # positonal things
      drive "diff"
)
```

3.2.2.5 - The Robot's Sensors

Now that Bigbob's body has been built let's move on to the sensors. We will put sonar and blobfinding sensors onto Bigbob so that it can detect walls and see coloured blobs it can interpret as rubbish to collect. We will also put a laser between Bigbob's teeth so that it can detect when an item passes in between them.

Bigbob's Sonar

We will start with the sonars. The first thing to do is to define a model for the sonar sensor that is going to be used on Bigbob:

```
define bigbobs_sonars sensor
(
      # parameters...
)
define bigbobs_ranger ranger
(
      # parameters...
)
```

Here we tell Player/Stage that we will define a type of sensor called bigbobs_sonars. Next, we'll tell Player/Stage to use these sensors in a ranging device. Let's put four sonars on Bigbob, one on the front of each tooth, and one on the front left and the front right corners of its body.

When building Bigbob's body we were able to use any location on a coordinate grid that we wanted and could declare our shape polygons to be any distance apart we wanted so long as we resized the model with `size`. In contrast, sensors - all sensors not just rangers - must be positioned according to the *robot's* origin and actual size. To work out the distances in metres it helps to do a drawing of where the sensors will go on the robot and their distances from the robot's origin. When we worked out the shape of Bigbob's body we used its actual size, so we can use the same drawings again to work out the distances of the sensors from the origin as shown in Figure 3.10.

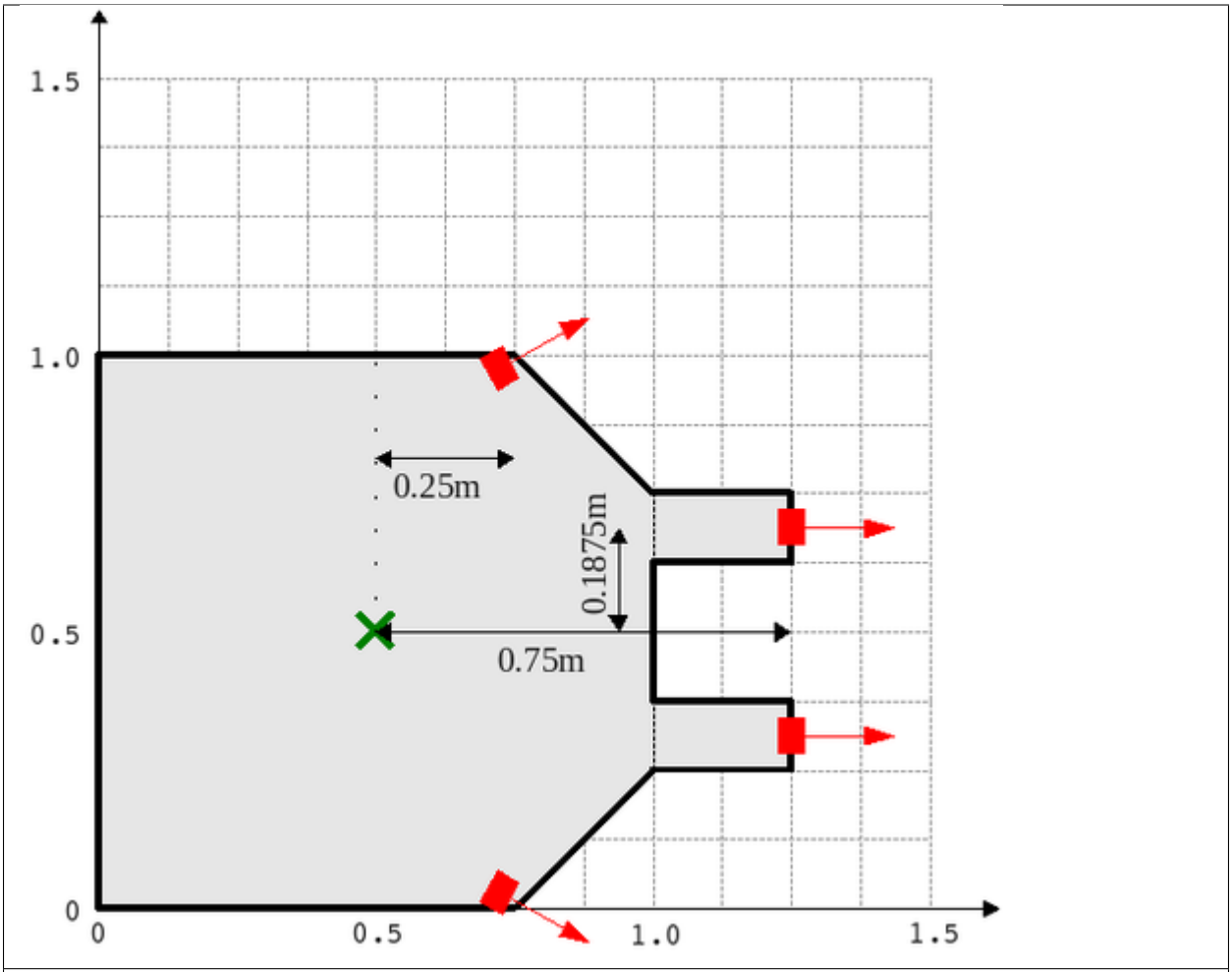

Figure 3.10: The position of Bigbob's sonars (in red) relative to its origin. The origin is marked with a cross, some of the distances from the origin to the sensors have been marked. The remaining distances can be done by inspection.

First, we'll define a single ranger (in this case sonar) sensor. To define the size, range and field of view of the sonars we just consult the sonar device's datasheet.

```
define bigbobs_sonar sensor
(
    # define the size of each transducer [xsize ysize zsize] in meters
    size [0.01 0.05 0.01 ]
    # define the range bounds [min max]
    range [0.3 2.0]
    # define the angular field of view in degrees
    fov 10
    # define the color that ranges are drawn in the gui
    color_rgba [ 0 1 0 1 ]
)
```

Then, define how the sensors are placed into the ranger device. The process of working out where the sensors go relative to the origin of the robot is the most complicated part of describing the sensor.

```
define bigbobs_sonars ranger
(
  # one line for each sonar [xpos ypos zpos heading]
  bigbobs_sonar( pose [ 0.75 0.1875 0 0]) # fr left tooth
  bigbobs_sonar( pose [ 0.75 -0.1875 0 0]) # fr right tooth
  bigbobs_sonar( pose [ 0.25 0.5 0 30]) # left corner
  bigbobs_sonar( pose [ 0.25 -0.5 0 -30]) # right corner
)
```

TRY IT OUT (driving a robot)

This file includes everything described up till now.

This will start player in the background, then start a "remote control" (also in the background). You may need to move the playerv window out of the way to see the Stage window.

See the playerv documentation for details on playerv. For now, the "remote control" just makes the ranger sensor cones appear.

Bigbob's Blobfinder

Now that Bigbob's sonars are done we will attach a blobfinder:

```
define bigbobs_eyes blobfinder
(
      # parameters
)
```

Bigbob is a rubbish-collector so here we should tell it what colour of rubbish to look for. Let's say that the intended application of Bigbob is in an orange juice factory and he picks up any stray oranges or juice cartons that fall on the floor. Oranges are orange, and juice cartons are (let's say) dark blue so Bigbob's blobfinder will look for these two colours:

```
define bigbobs_eyes blobfinder
(
      # number of colours to look for
      colors_count 2

      # which colours to look for
      colors ["orange" "DarkBlue"]
)
```

Then we define the properties of the camera, again these come from a datasheet:

```
define bigbobs_eyes blobfinder
(
      # number of colours to look for
      colors_count 2

      # which colours to look for
      colors ["orange" "DarkBlue"]

      # camera parameters
      image [160 120]   #resolution
```

(continues on next page)

(continued from previous page)

```
      range 5.00          # m
      fov 60              # degrees
)
```

TRY IT OUT (blobfinder)

Similar to the previous example, `playerv` just makes the camera show up in the PlayerViewer window.

Bigbob's Laser

The last sensor that needs adding to Bigbob is the laser, which will be used to detect whenever a piece of rubbish has been collected, the laser's location on the robot is shown in Figure 3.11. Following the same principles as for our previous sensor models we can create a description of this laser:

```
define bigbobs_laser sensor
(
      size [0.025 0.025 0.025]
      range [0 0.25]              # max = dist between teeth in m
      fov 20                      # does not need to be big
      color_rgba [ 1 0 0 0.5]
      samples 180                 # number of ranges measured
)
define bigbobs_lasers ranger
(
      bigbobs_laser( pose [ 0.625 0.125 -0.975 270 ])
)
```

With this laser we've set its maximum range to be the distance between teeth, and the field of view is arbitrarily set to *20* degrees. We have calculated the laser's `pose` in exactly the same way as the sonars `pose`, by measuring the distance from the laser's centre to the robot's origin (which we set with the `origin` parameter earlier). The z coordinate of the pose parameter when describing parts of the robot is relative to the very top of the robot. In this case the robot is 1 metre tall so we put the laser at *-0.975* so that it is on the ground. The laser's yaw is set to *270* degrees so that it points across Bigbob's teeth. We also set the size of the laser to be 2.5cm cube so that it doesn't obstruct the gap between Bigbob's teeth.

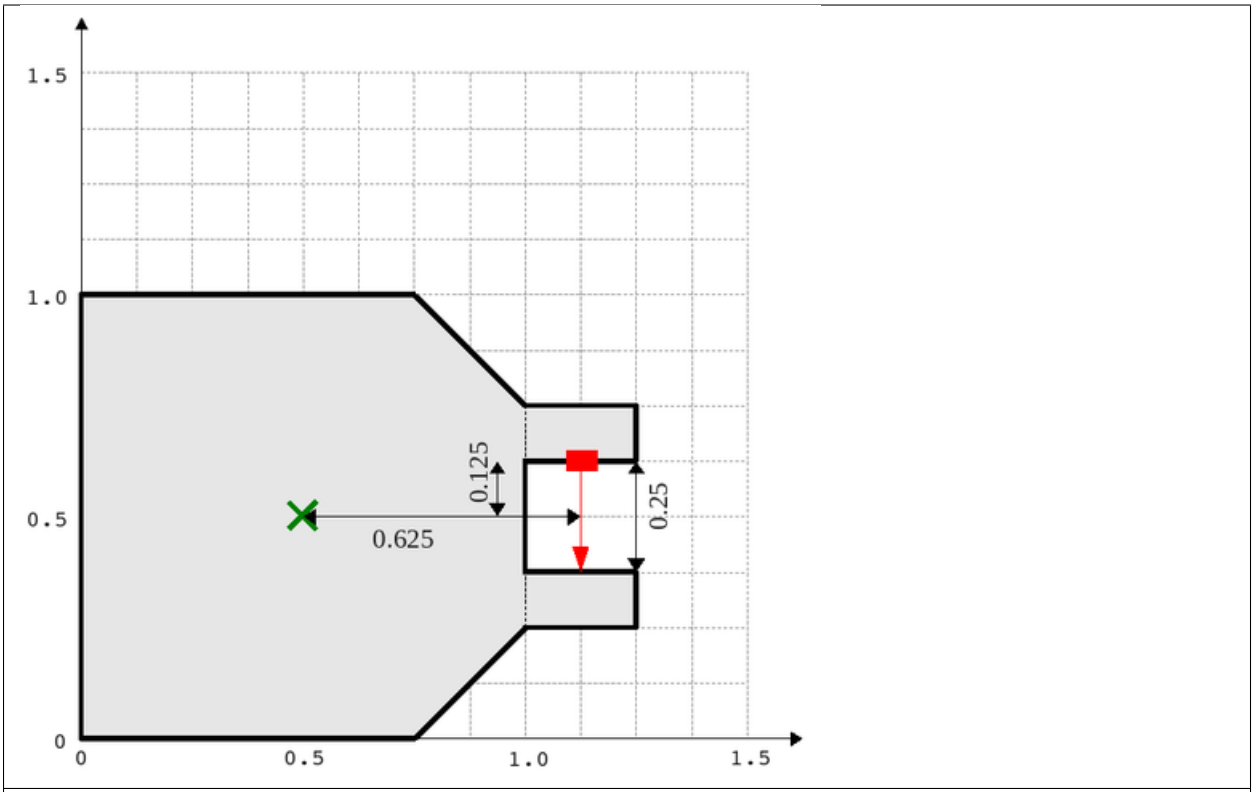

Figure 3.11: The position of Bigbob's laser (in red) and its distance, in metres, relative to its origin (marked with a cross).

Now that we have a robot body and sensor models all we need to do is put them together and place them in the world. To add the sensors to the body we need to go back to the `bigbob position` model:

```
define bigbob position
(
      # actual size
      size [1.25 1 1]
      # centre of rotation offset
      origin [0.125 0 0]

      # the shape of Bigbob
      block
            ...
            ...
            ...

      # positonal things
      drive "diff"

      # sensors attached to bigbob
      bigbobs_sonars()
      bigbobs_eyes()
      bigbobs_laser()
)
```

The extra line `bigbobs_sonars()` adds the sonar model called `bigbobs_sonars()` onto the `bigbob` model, likewise for `bigbobs_eyes()` and `bigbobs_laser()`.

At this point it's worthwhile to copy this into a .inc file, so that the model could be used again in other simulations or worlds. This file can also be found in the example code in /Ch5.3/bigbob.inc

To put our Bigbob model into our empty world (see *Section 3.1.3 - Making a Basic Worldfile*) we need to add the robot to our worldfile `empty.world`:

```
include "map.inc"
include "bigbob.inc"

# size of the whole simulation
size [15 15]


# configure the GUI window
window
(
      size [ 700.000 700.000 ]
      scale 35
)


# load an environment bitmap
floorplan
(
      bitmap "bitmaps/cave.png"
      size [15 15 0.5]
)

bigbob
(
      name "bob1"
      pose [-5 -6 0 45]
      color "green"
)
```

Here we've put all the stuff that describes Bigbob into a .inc file `bigbob.inc`, and when we include this, all the code from the .inc file is inserted into the .world file. The section here is where we put a version of the bigbob model into our world:

```
bigbob
(
      name "bob1"
      pose [-5 -6 0 45]
      color "green"
)
```

Bigbob is a model description, by not including any `define` stuff in the top line there it means that we are making an instantiation of that model, with the name `bob1`. Using an object-oriented programming analogy, `bigbob` is our class, and `bob1` is our object of class `bigbob`. The `pose [x y yaw]` parameter works in the same way as `spose [x y yaw]` does. The only differences are that the coordinates use the centre of the simulation as a reference point and `pose` lets us specify the initial position and heading of the entire `bob1` model, not just one sensor within that model.

Finally we specify what colour `bob1` should be, by default this is red. The `pose` and `color` parameters could have been specified in our bigbob model but by leaving them out it allows us to vary the colour and position of the robots for each different robot of type `bigbob`, so we could declare multiple robots which are the same size, shape and have the same sensors, but are rendered by Player/Stage in different colours and are initialised at different points in the map.

When we run the new `bigbob6.world` with Player/Stage we see our Bigbob robot is occupying the world, as shown

in Figure 3.12.

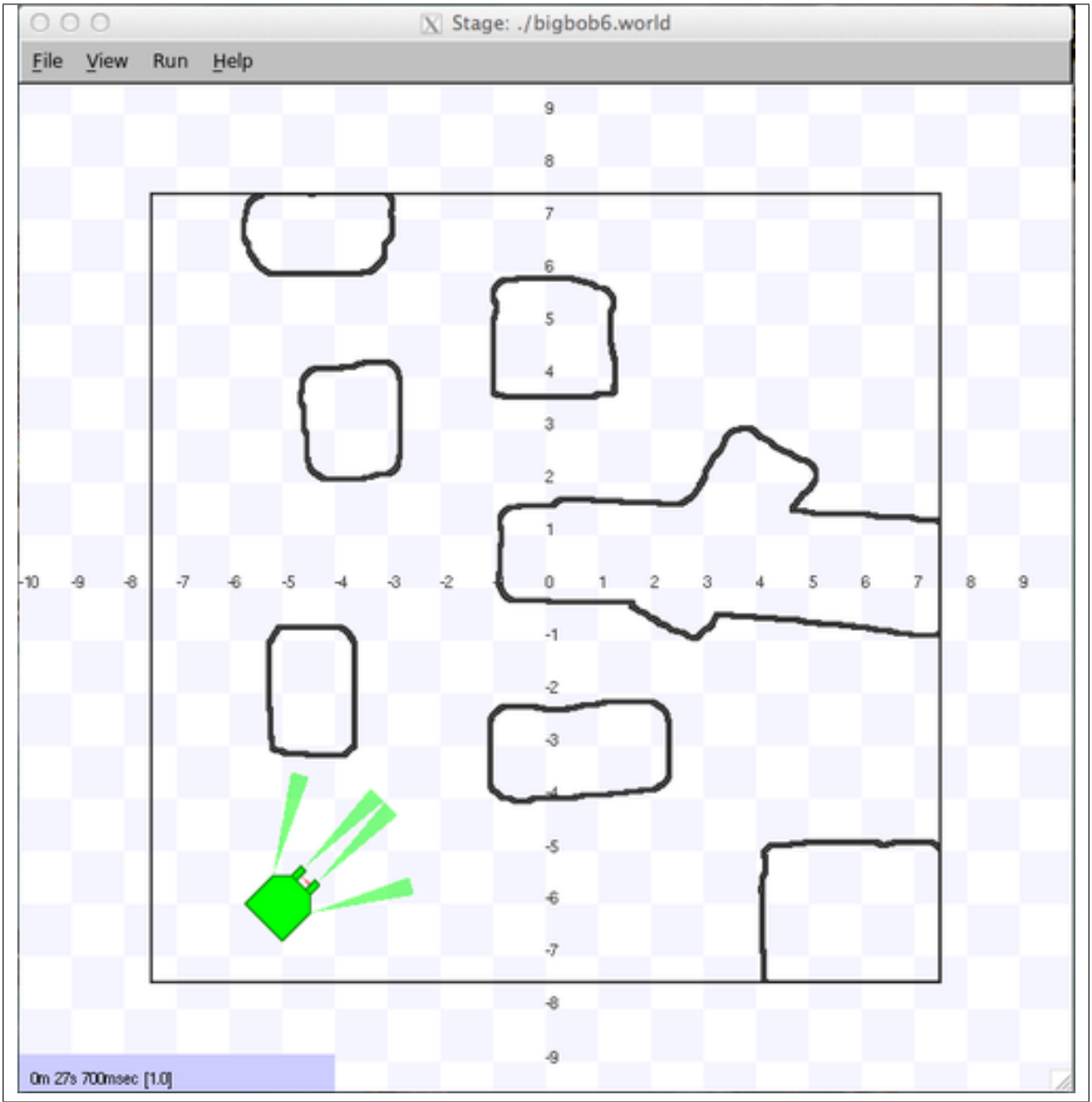

Figure 3.12: Our bob1 robot placed in the simple world, showing the range and field of view of all of the ranger sensors.

TRY IT OUT (Bigbob in environment)

This should show you Figure 3.12

You may wish to zoom in on the teeth to see the tooth laser.

3.2.3 - Building Other Stuff

We established in *Section 3.2.2 - An Example Robot* that Bigbob works in a orange juice factory collecting oranges and juice cartons. Now we need to build models to represent the oranges and juice cartons so that Bigbob can interact with things.

oranges

We'll start by building a model of an orange:

```
define orange model
(
      # parameters...
)
```

The first thing to define is the shape of the orange. The `block` parameter is one way of doing this, which we can use to build a blocky approximation of a circle. An alternative to this is to use `bitmap` which we previously saw being used to create a map. What the bitmap command actually does is take in a picture, and turn it into a series of blocks which are connected together to make a model the same shape as the picture, as illustrated in Figure 3.13 for an alien bitmap.

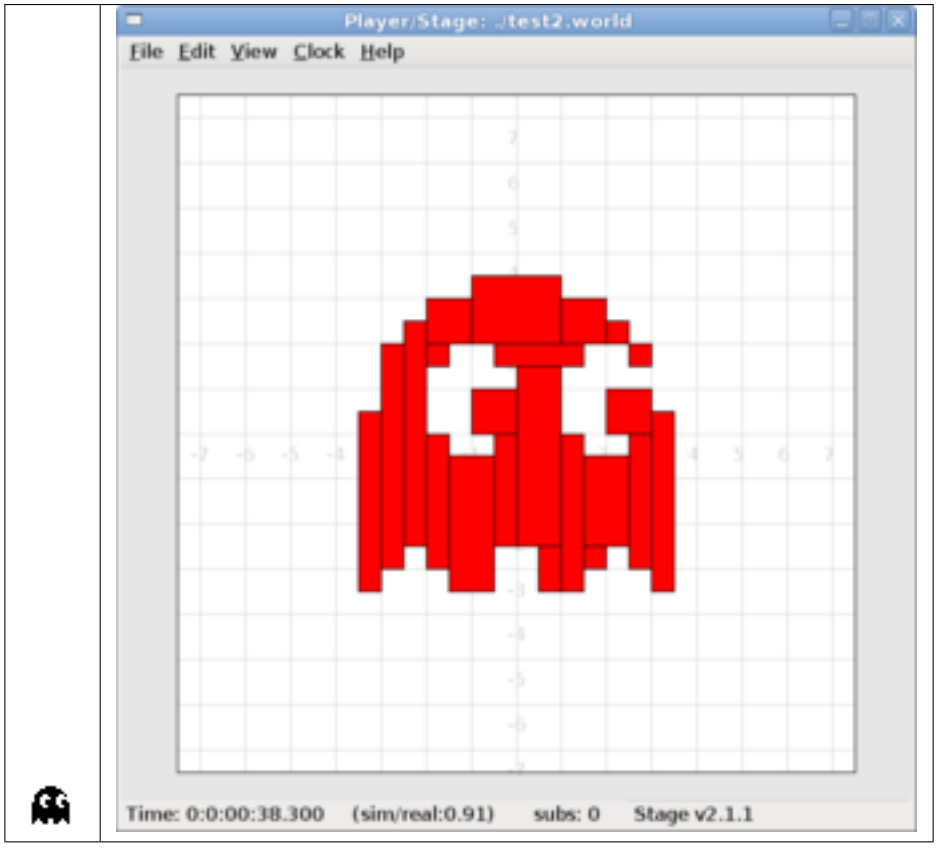

Figure 3.13: The left image is the original picture, the right image is its Stage interpretation.

In our code, we don't want an alien, we want a simple circular shape (see Figure 3.14), so we'll point to a circular bitmap.

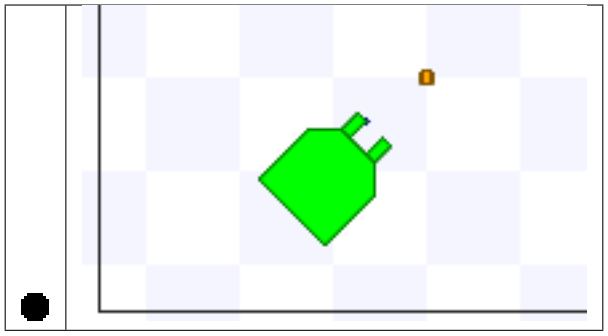

Figure 3.14: The orange model rendered in the same Stage window as Bigbob.

```
define orange model
(
      bitmap "bitmaps/circle.png"
      size [0.15 0.15 0.15]
      color "orange"
)
```

In this bit of code we describe a model called `orange` which uses a bitmap to define its shape and represents an object which is *15cm* x *15cm* x *15cm* and is coloured orange. Figure 3.14 shows our orange model next to Bigbob.

Juice Cartons

Building a juice carton model is similarly quite easy:

```
define carton model
(
      # a carton is retangular
      # so make a square shape and use size[]
      block
      (
            points 4
            point[0] [1 0]
            point[1] [1 1]
            point[2] [0 1]
            point[3] [0 0]
            z [0 1]
      )

      # average litre carton size is ~ 20cm x 10cm x 5cm ish
      size [0.1 0.2 0.2]

      color "DarkBlue"
)
```

We can use the `block` command since juice cartons are boxy, with boxy things it's slightly easier to describe the shape with `block` than drawing a bitmap and using that. In the above code I used `block` to describe a metre cube (since that's something that can be done pretty easily without needing to draw a carton on a grid) and then resized it to the size I wanted using `size`.

Putting objects into the world

Now that we have described basic `orange` and `carton` models it's time to put some oranges and cartons into the simulation. This is done in the same way as our example robot was put into the world:

```
orange
(
      name "orange1"
      pose [-2 -5 0 0]
)

carton
(
      name "carton1"
      pose [-3 -5 0 0]
)
```

We created models of oranges and cartons, and now we are declaring that there will be an instance of these models (called `orange1` and `carton1` respectively) at the given positions. Unlike with the robot, we declared the `color` of the models in the description so we don't need to do that here. If we did have different colours for each orange or carton then it would mess up the blobfinding on Bigbob because the robot is only searching for orange and dark blue. At this point it would be useful if we could have more than just one orange or carton in the world (Bigbob would not be very busy if there wasn't much to pick up), it turns out that this is also pretty easy:

```
orange(name "orange1" pose [-1 -5 0 0])
orange(name "orange2" pose [-2 -5 0 0])
orange(name "orange3" pose [-3 -5 0 0])
orange(name "orange4" pose [-4 -5 0 0])

carton(name "carton1" pose [-2 -4 0 0])
carton(name "carton2" pose [-2 -3 0 0])
carton(name "carton3" pose [-2 -2 0 0])
carton(name "carton4" pose [-2 -1 0 0])
```

Up until now we have been describing models with each parameter on a new line, this is just a way of making it more readable for the programmer – especially if there are a lot of parameters. If there are only a few parameters or you want to be able to comment it out easily, it can all be put onto one line. Here we declare that there will be four `orange` models in the simulation with the names `orange1` to `orange4`, we also need to specify different poses for the models so they aren't all on top of each other. Properties that the orange models have in common (such as shape, colour or size) should all be in the model definition.

TRY IT OUT (full worldfile)

This should show you Figure 3.15.

The full worldfile is at `<source_code>/Ch3/bigbob7.world`, this includes the orange and carton models as well as the code for putting them in the simulation. Figure 3.15 shows the populated Player/Stage simulation.

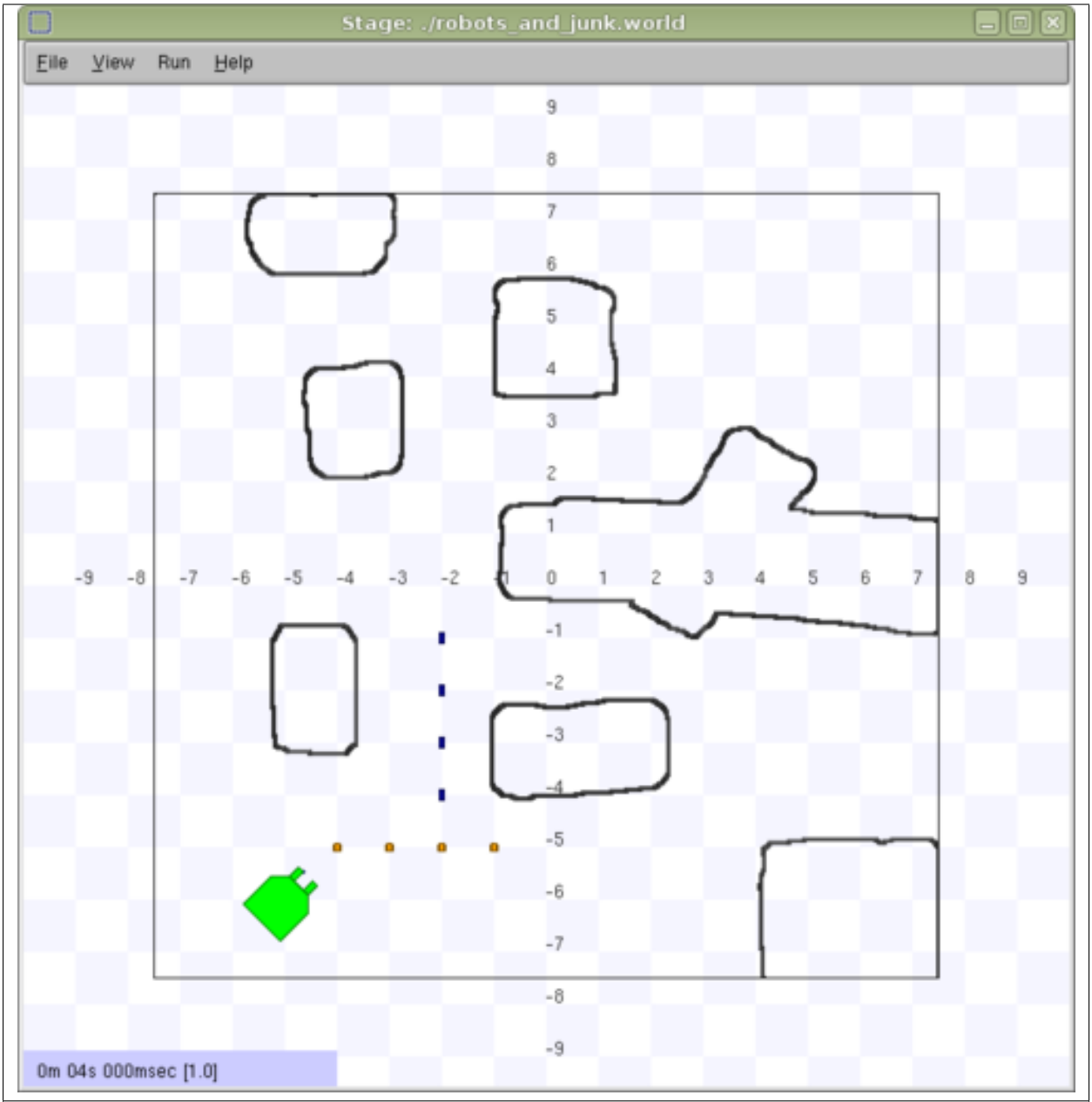

Figure 3.15: The Bigbob robot placed in the simulation along with junk for it to pick up.

Fig. 2: img

1.4 Donnie Robot

nao mostrar como montar e configurar um robo. para isso, aponte para o manual do desenvolvedor. nesta secao assume-se que o robo esta pronto para uso.

1.4.1 Power Up

procedimentos de inicialização

1.4.2 Running GoDonnie

como executar o GoDonnie com robô fisico

1.4.3 Shutting Down

como desligar o robo

1.5 Donnie Vibrating Belt

With the goal of improving the quality of life of people with visual impairment and helping their mobility through a better perception of the environment it was created a tactile belt. Capable of working with different intensities, the tactile belt makes the user perceive, through vibrations, the approaching of an object. Initially the belt counted on a Kinect sensor to "see" the environment around and to identify obstacles. It also had 35 engines that covered the abdomen, vibrating as information was received. Now the belt was adapted and implanted in the Donnie Project and has 14 engines. It has four types of vibration and it's connected with the Donnie robot and with the simulation robot. The belt allows the person to perceive through vibrations the distance of the objects around the robot.

1.5.1 Power Up

procedimentos de inicialização

1.5.2 Running the Belt

como executar o cinto

1.5.3 Shutting Down

como desligar o cinto

1.6 Donnie Contributors

The list of contributors to this document.

- @Alexandre Amory
- @Roger
- @Renan
- @Marcelo
- @Davi
- @Beltrano com webpage

CHAPTER 2

Papers

If you are using Donnie and/or its software on your research projects, please cite our papers:

```
@inproceedings{oliveira2017teaching,
  title={Teaching Robot Programming Activities for Visually Impaired Students: A␣
↪Systematic Review},
  author={Oliveira, Juliana Damasio and de Borba Campos, M{\'a}rcia and de Morais␣
↪Amory, Alexandre and Manssour, Isabel Harb},
  booktitle={International Conference on Universal Access in Human-Computer␣
↪Interaction},
  pages={155--167},
  year={2017},
  organization={Springer}
}
```

```
@inproceedings{guilherme2017donnie,
  title={Donnie Robot: Towards an Accessible And Educational Robot for Visually␣
↪Impaired People},
  author={Guilherme H. M. Marques, Daniel C. Einloft, Augusto C. P. Bergamin, Joice A.
↪ Marek, Renan G. Maidana Marcia B. Campos, Isabel H. Manssour, Alexandre M. Amory},
  booktitle={Latin American Robotics Symposium (LARS)},
  year={2017}
}
```

CHAPTER 3

Disclaimer

Donnie and its software are protected under the MIT License:

CHAPTER 4

Feedback

Don't hesitate to ask about additional info or the next guides, and also if you find some mistakes, please let us know. Issues and push requests can be done on github.