How to Use Player/Stage

Jennifer Owen

April 15, 2010

This document is intended as a guide for anyone learning Player/Stage for the first time. It explains the process of setting up a new simulation environment and how to then make your simulation do something, using a case study along the way. Whilst it is aimed at Player/Stage users, those just wishing to use Player on their robot may also find sections of this document useful (particularly the parts about coding with Player).

If you have any questions about using Player/Stage there is a guide to getting help from the Player comunity here:

`http://playerstage.sourceforge.net/wiki/Getting_help`

This edition of the manual uses Stage version 3.2.X as there are significant differences with the previous versions of Stage and the previous edition of this manual is now out of date. If you find any problems or errors *in this manual* then please do email me, although for help with Player/Stage I would strongly recommend the above link.

The code listings from this document can be downloaded from `www.jenny-owen.co.uk`, plus a *small* amount of supplementary material. The official Player and Stage manuals are, however, much more helpful.

Contents

Chapter 1

Introduction

Player/Stage is a robot simulation tool, it comprises of one program, Player, which is a *Hardware Abstraction Layer*. That means that it talks to the bits of hardware on the robot (like a claw or a camera) and lets you control them with your code, meaning you don't need to worry about how the various parts of the robot work. Stage is a plugin to Player which listens to what Player is telling it to do and turns these instructions into a simulation of your robot. It also simulates sensor data and sends this to Player which in turn makes the sensor data available to your code.

A simulation then, is composed of three parts:

- Your code. This talks to Player.
- Player. This takes your code and sends instructions to a robot. From the robot it gets sensor data and sends it to your code.
- Stage. Stage interfaces with Player in the same way as a robot's hardware would. It receives instructions from Player and moves a simulated robot in a simulated world, it gets sensor data from the robot in the simulation and sends this to Player.

Together Player and Stage are called Player/Stage, and they make a simulation of your robots.

These instructions will be focussing on how to use Player/Stage to make a simulation, but hopefully this will still be a useful resource for anyone just using Player (which is the same thing but on a real robot, without any simulation software).

1.1 A Note on Installing Player/Stage

Instructions on how to install Player/Stage onto your computer aren't really the focus of this document. It is very difficult though. If you're lucky the install will work first time but there are a lot of dependencies which may need installing. For computers running Ubuntu there is a very good set of instructions here (including a script for downloading the many prerequisites):

`http://www.control.aau.dk/~tb/wiki/index.php/Installing_Player_and_Stage_in_Ubuntu`

For MAC users you might find the following install instructions useful:

`http://alanwinfield.blogspot.com/2009/07/installing-playerstage-on-os-x-with.html`

Alternatively, you could try the suggestions on the Player "getting help" page:

`http://playerstage.sourceforge.net/wiki/Getting_help`

Chapter 2

The Basics

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[1], 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.

[1]Or you can download or write your own drivers, but I'm not going to talk about how to do this here.

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".

- 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[2]. They specify the syntax and semantics of how drivers and Player interact.

- 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 3 things quite well with an example.

> Consider the laser 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 laser 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 laser interface.
>
> The sicklms200 driver can be bound to the laser interface ... to create a device, which might have the following address:
>
> localhost:6665:laser: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 laser, 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

[2] http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group__interfaces.html

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 section 4.

Chapter 3

Building a World

First we will run a world and configuration file that comes bundled with Stage. In your bash shell navigate to the Stage/worlds folder, by default (in Linux at least) this is /usr/local/share/stage/worlds. Once in the correct folder type the following command to run the "simple world" that comes with Player/Stage:

```
player simple.cfg
```

Assuming Player/Stage is installed properly you should now have a window open which looks figure 3.1.

Congratulations, you can now build Player/Stage simulations! You may note that the robot in the simple.cfg simulation will immediately start moving, don't worry about this for now, we will discuss how to achieve this in section 3.2.

3.1 Building an Empty World

As you can see in section 3, 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` 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 section 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

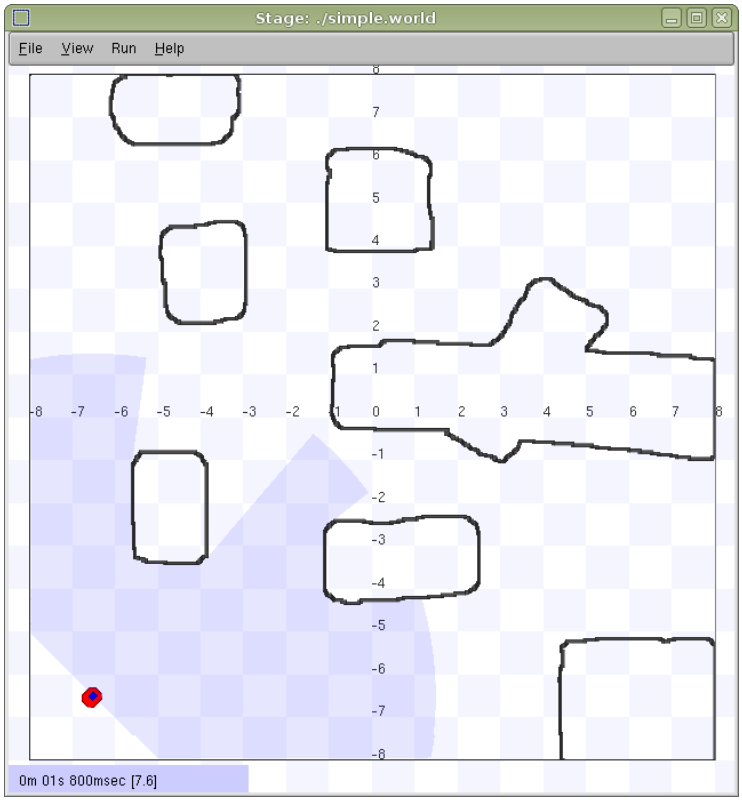

Figure 3.1: The simple.cfg world after being run

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

  gui_nose 0
  gui_grid 0
  gui_move 0
  gui_outline 0
  gripper_return 0
  fiducial_return 0
  laser_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 section 3 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 3.2.1. Setting `fiducial_return` to 0 means that the map cannot be detected by a fiducial 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.

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"
   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

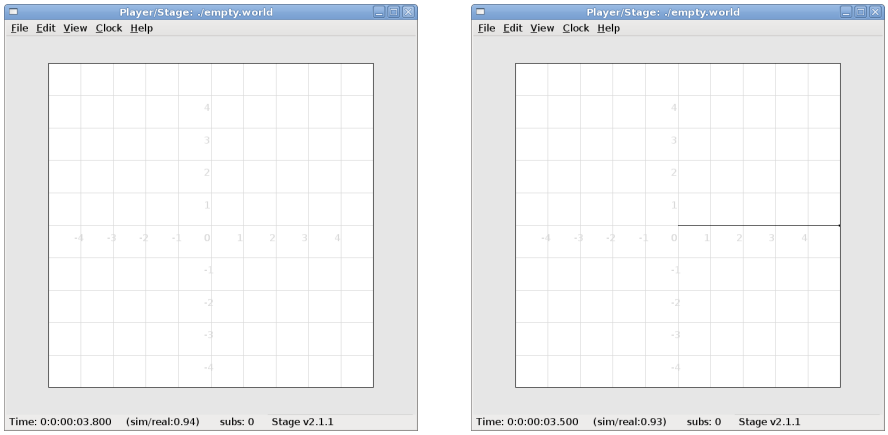

Figure 3.2: The left picture shows an empty map without a nose. The right picture shows the same map with a nose to indicate orientation, this is the horizontal line from the centre of the map to the right, it shows that the map is actually facing to the right.

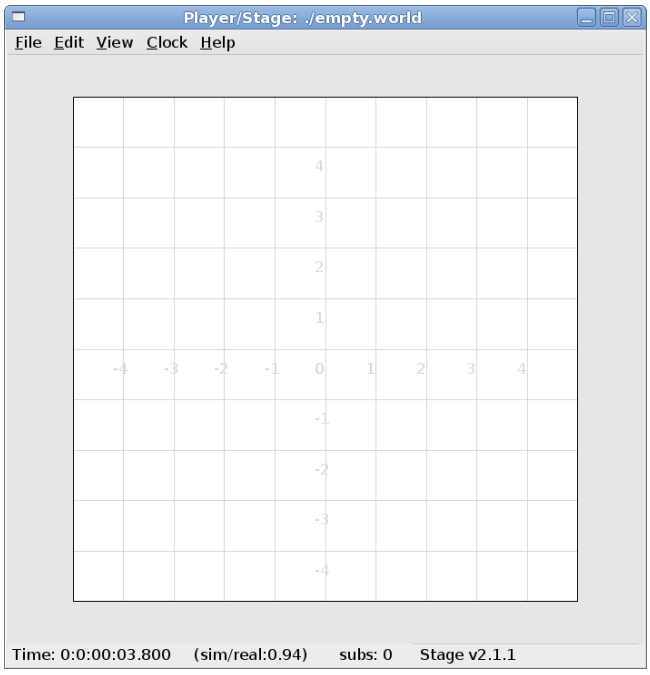

Figure 3.3: An empty map with gui_grid enabled. With gui_grid disabled this would just be an empty white square.

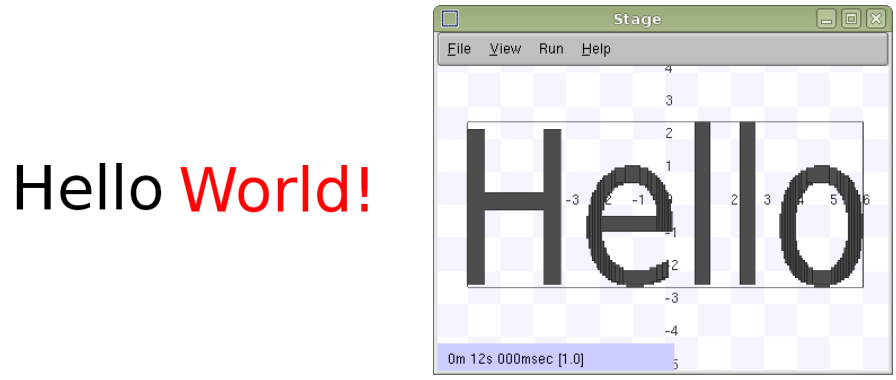

Figure 3.4: The left image is our "helloworld.png" bitmap, the right image is what Player/Stage interprets that bitmap as. The coloured areas are walls, the robot can move everywhere else.

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.

A full list of model parameters and their descriptions can be found in the official Stage manual[1]. 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.

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:

[1] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model.html

```
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 $\frac{window\_size}{floorplan\_size}$ 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"[2].

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[3].
Finally, we are able to write a worldfile!

```
include "map.inc"

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

[2] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__worldgui.html
[3] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__world.html

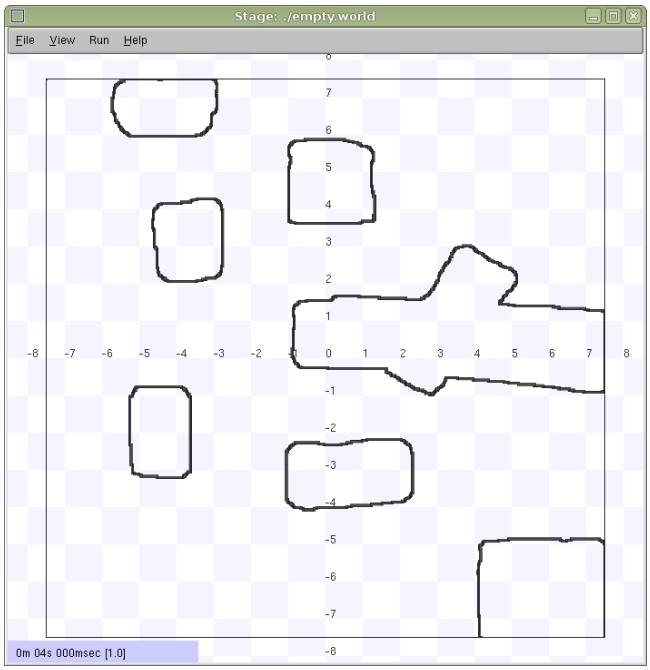

Figure 3.5: Our Empty World.

```
# 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) to get the simulation shown in figure 3.5.

3.2 Building a Robot

In Player/Stage a robot is just a slightly advanced kind of model, all the parameters described in section 3.1.1 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) 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[4] although only the following are supported by the current distribution of Stage (version 3.2.X). Unless otherwise stated, these models use the Player interface that shares its name:

camera

[5] 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]`: the horizontal angle the camera can move through (pan) and the vertical angle (tilt). So for instance `pantilt [90 20]` allows the camera to move 45° left and 45° right and 10° up and 10° down.

blobfinder

[6] This 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. The blobfinder can only find a model if its `blob_return` parameter is true. 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.[7]
- `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 RADIANS*.

[4] http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group__interfaces.html
[5] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model__camera.html
[6] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model__blobfinder.html
[7] rgb.txt can normally be found at /usr/share/X11/rgb.txt assuming it's properly installed, alternatively a Google search for "rgb.txt" will give you the document.

fiducial

[8] 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_key`s 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 RADIANS*.

ranger

[9] This simulates any kind of obstacle detection device (e.g. sonars or infrared sensors). These can locate models whose `ranger_return` is true. Using a ranger model you can define any number of ranger devices and apply them all to a single robot. Unlike the other types of model this doesn't use the interface with its name but instead the `sonar` interface, there is more about this in section 4.2. The parameters for the `ranger` model and their inputs are described in the Stage manual, but basically:

- `scount <int>`: The number of ranger sensors in this ranger model
- `spose[ranger_number] [x y yaw]`: Tells the simulator where the rangers are placed around the robot. How to write the `[x y yaw]` data is explained in section 3.2.2.
- `ssize [x y]`: how big the sensors are.
- `sview [min max fov]`: defines the maximum and minimum distances that can be sensed and also the field of view *in DEGREES*.

laser

[10] A laser is a special case of ranger sensor which only allows one ranger (so there's none of the `scount`, `spose` stuff), but it has a very large field of view. If a model has its `laser_return` parameter enabled then a laser can detect it. Details about laser parameters can be found in the Stage manual, however the most useful parameters are:

[8] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model__fiducial.html
[9] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model__ranger.html
[10] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model__laser.html

- `samples`: The number of ranger readings the laser 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.
- `range_max`: the maximum range of the laser.
- `fov`: the field of view of the laser *in RADIANS*.

gripper

[11] The gripper model is a simulation of the gripper you get on a Pioneer robot.[12] 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 3.2.2.

position

[13] 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.

[11] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model__gripper.html

[12] The Pioneer grippers looks like a big block on the front of the robot with two big sliders that close around an object.

[13] http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model__position.html

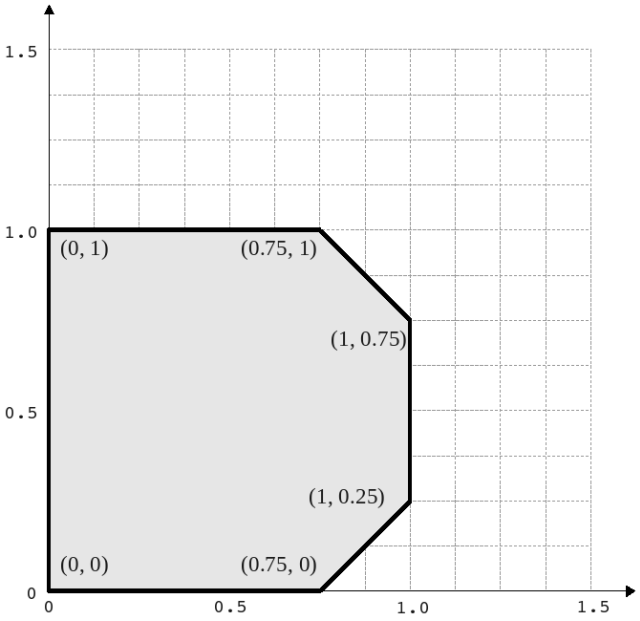

Figure 3.6: The basic shape we want to make Bigbob, the units on the axes are in metres.

- `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.

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[14]:

```
define bigbob position

      block
      (
            points 6
```

[14]In this example we're using blocks with the position model type but we could equally use it with other model types.

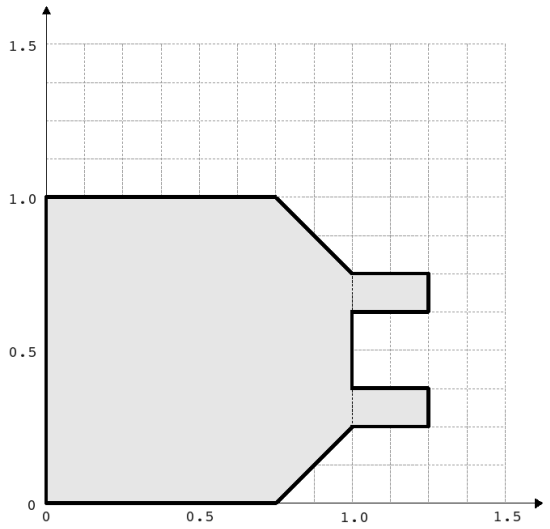

Figure 3.7: The new shape of Bigbob.

```
            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]
      )
)
```

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. It is important to go around the robot doing each corner in turn going *anti-clockwise* around the robot, otherwise Player/Stage won't properly render the block. 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]`. 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:

```
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]
              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 1 x 1 x 1 metres, 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.
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

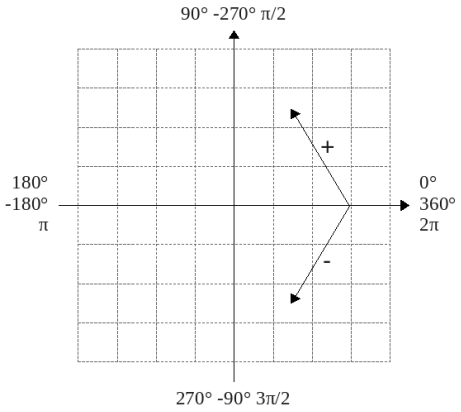

Figure 3.8: A cartesian grid showing how angles are described.

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°. This is also why in section 3.1 the `gui_nose` shows the map is facing to the right. Figure 3.9 shows a few examples of robots with different yaws.

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
            ...
            ...
            ...
)
```

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

```
define bigbob position
```

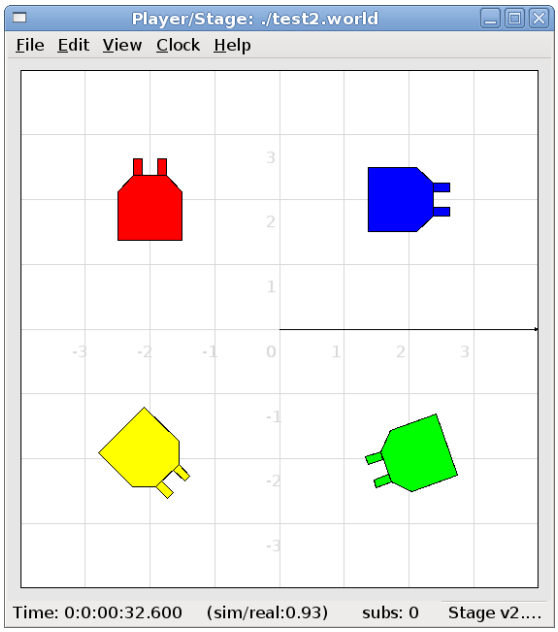

Figure 3.9: Starting from the top right robot and working anti-clockwise, the yaws of these robots are 0, 90, -45 and 200.

```
(
      # 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"
)
```

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.

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

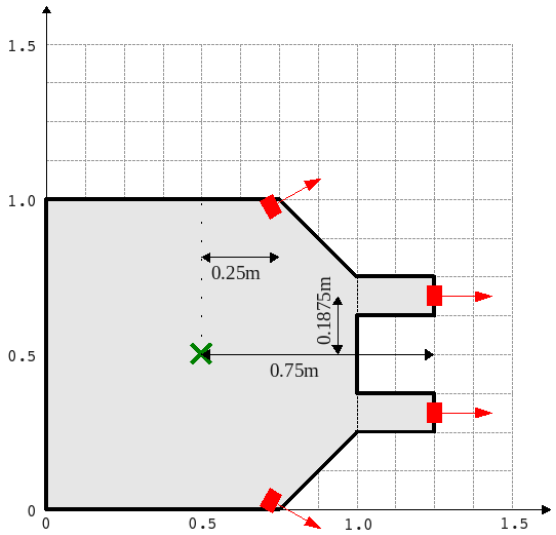

Figure 3.10: The position of Bigbob's sonars (in red) relative to its origin. The origin is marked with a cross, some of the distances from the origin to the sensors have been marked. The remaining distances can be done by inspection.

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

Here we tell Player/Stage that we will `define` a set of sonar sensors called `bigbobs_sonars` and we're using the model type `ranger` to tell Player/Stage that this is a model of some ranging devices. 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.

Now we know how many sensors we want, and their coordinates relative to the origin we can begin to build our model of the sonar array. Here we will use the `scount` and `spose` parameters mentioned in 3.2.1. The values for `spose` are `[x y yaw]`, remember that yaw is in degrees and is measured relative to the x axis.

```
define bigbobs_sonars ranger
(
      # number of sonars
```

```
      scount 4

      # define the pose of each transducer [xpos ypos heading]
      spose[0] [ 0.75 0.1875 0 ]     #fr left tooth
      spose[1] [ 0.75 -0.1875 0 ]    #fr right tooth
      spose[2] [ 0.25 0.5 30]        # left corner
      spose[3] [ 0.25 -0.5 -30]      # right corner

)
```

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, the rest is easy. To define the size, range and field of view of the sonars we just consult the sonar device's datasheet.

```
define bigbobs_sonars ranger
(
      # number of sonars
      scount 4

      # define the pose of each transducer [xpos ypos heading]
      spose[0] [ 0.75 0.1875 0 ]     #fr left tooth
      spose[1] [ 0.75 -0.1875 0 ]    #fr right tooth
      spose[2] [ 0.25 0.5 30]        # left corner
      spose[3] [ 0.25 -0.5 -30]      # right corner

      # define the field of view of each transducer
      # [range_min range_max view_angle]
      sview [0.3 2.0 10]

      # define the size of each transducer [xsize ysize] in metres
      ssize [0.01 0.05]
)
```

Now that Bigbob's sonars are done we will attached 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
      range 5.00
      fov 1.047196667 # 60 degrees = pi/3 radians
)
```

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 laser
(
      # distance between teeth in metres
      range_max 0.25

      # does not need to be big
      fov 20

      pose [0.625 0.125 -0.975 270]
      size [0.025 0.025 0.025]
)
```

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°. We have calculated the laser's `pose` in exactly the same way as the sonars `spose`, 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° 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.

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
```

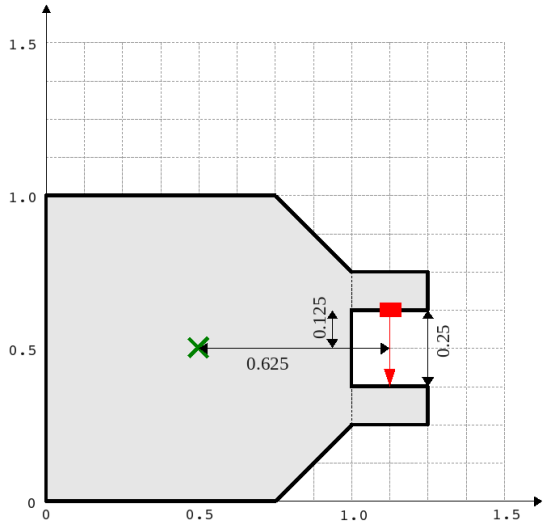

Figure 3.11: The position of Bigbob's laser (in red) and its distance, in metres, relative to its origin (marked with a cross).

```
(
      # 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()`. After this final step we now have a complete model of our robot bigbob, the full code for which can be found in appendix A. 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.

To put our Bigbob model into our empty world (see section 3.1.3) 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 was 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

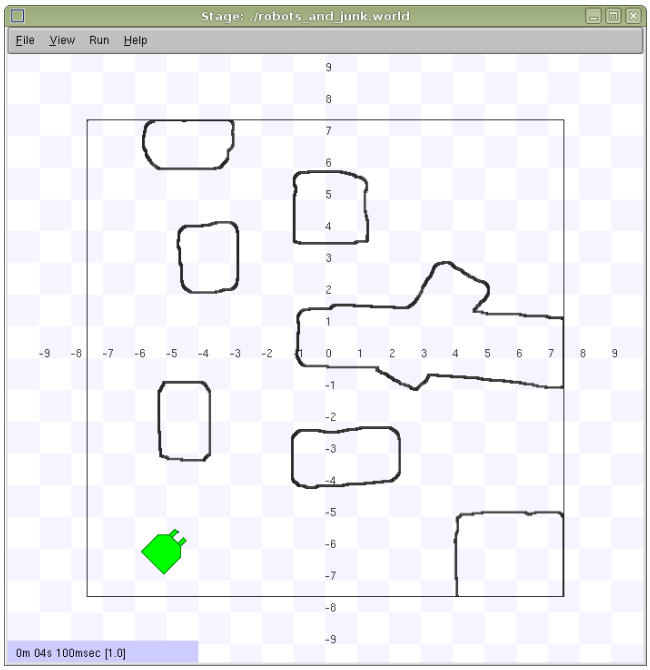

Figure 3.12: Our bob1 robot placed in the empty world.

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 empty.world with Player/Stage we see our Bigbob robot is occupying the world, as shown in figure 3.12.

3.3 Building Other Stuff

We established in section 3.2.2 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.

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. This is illustrated in figure 3.13.

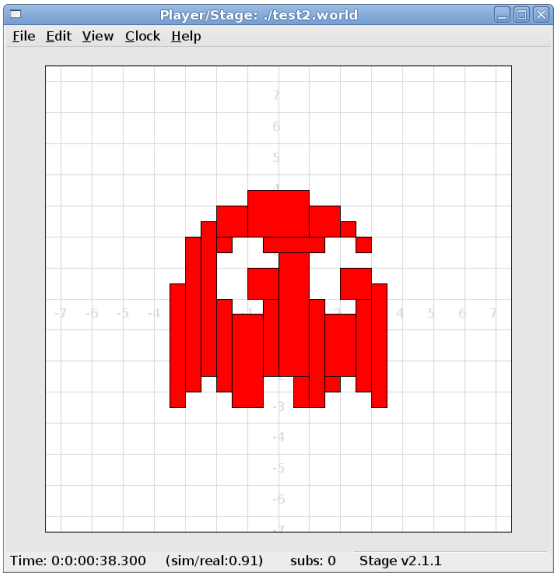

Figure 3.13: The left image is the original picture, the right image is its Player/Stage interpretation.

Figure 3.14: ./bitmaps/circle.png

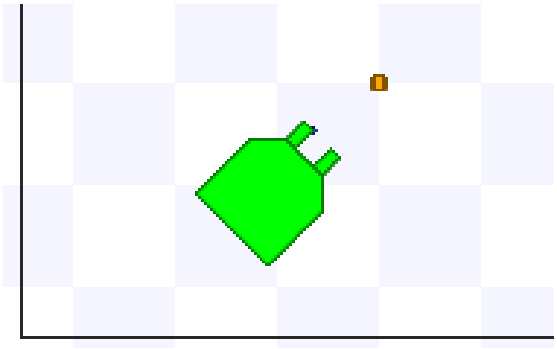

Figure 3.15: The orange model rendered in the same Player/Stage window as Bigbob.

```
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 3.15 shows our orange model next to Bigbob.

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`.

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.

The full worldfile is included in appendix B, this includes the orange and carton models as well as the code for putting them in the simulation. Figure 3.16 shows the populated Player/Stage simulation.

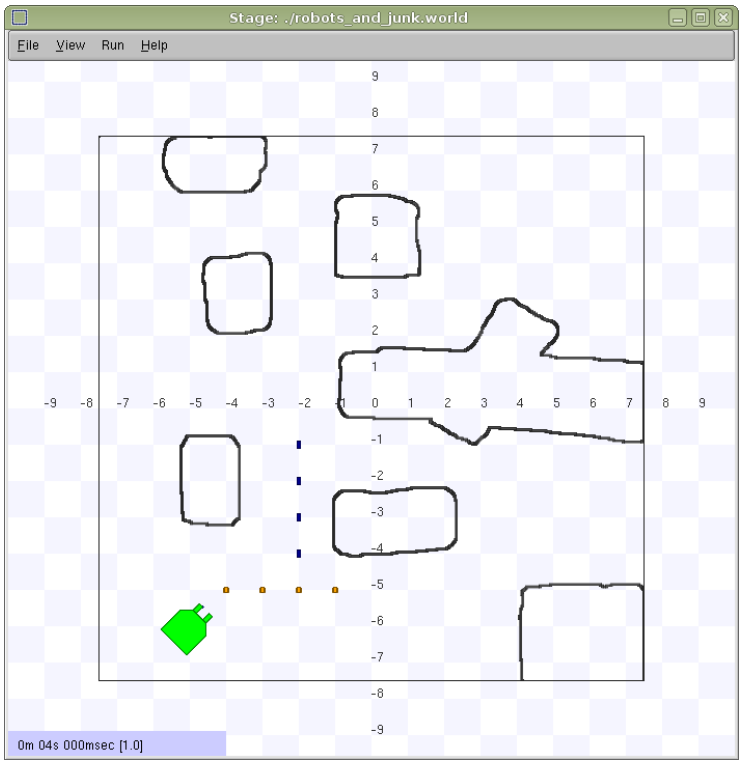

Figure 3.16: The Bigbob robot placed in the simulation along with junk for it to pick up.

Chapter 4

Writing a Configuration (.cfg) File

As mentioned earlier, Player is a hardware abstraction layer which connects your code to the robot's hardware. It does this by acting as a Server/Client type program where your code and the robot's sensors are clients to a Player server which then passes the data and instructions around to where it all needs to go. This stuff will be properly explained in section 5, it all sounds more complicated than it is because Player/Stage takes care of all the difficult stuff. The configuration file is needed in order to tell the Player server which drivers to use and which interfaces the drivers will be using.

For each model in the simulation or device on the robot that you want to interact with, you will need to specify a driver. This is far easier than writing worldfile information, and follows the same general syntax. The driver specification is in the form:

```
driver
(
      name "driver_name"
      provides [device_address]
      # other parameters...
)
```

The `name` and `provides` parameters are mandatory information, without them Player won't know which driver to use (given by `name`) and what kind of information is coming from the driver (`provides`). The `name` parameter is not arbitrary, it must be the name of one of Player's inbuilt drivers[1] that have been written for Player to interact with a robot device. A list of supported driver names is in the Player Manual[2], although when using Stage the only one that is needed is `"stage"`.

The `provides` parameter is a little more complicated than `name`. It is here that you tell Player what interface to use in order to interpret information given out by the driver (often this is sensor information from a robot), any information

[1] It is also possible to build your own drivers for a hardware device but this document won't go into how to do this because it's not relevant to Player/Stage.

[2] http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group__drivers.html

that a driver `provides` can be used by your code. For a Stage simulated robot the `"stage"` driver can provide the interfaces to the sensors discussed in section 3.2.1. Each interface shares the same name as the sensor model, so for example a `ranger` model would use the `ranger` interface to interact with Player and so on (the only exception to this being the `position` model which uses the `position2d` interface). The Player manual contains a list of all the different interfaces that can be used[3], the most useful ones have already been mentioned in section 3.2.1, although there are others too numerable to list here.

The input to the `provides` parameter is a "device address", which specifies which TCP port an interface to a robot device can be found, section 4.1 has more information about device addresses. This uses the key:host:robot:interface:index form separated by white space.

```
provides ["key:host:robot:interface:index"
          "key:host:robot:interface:index"
          "key:host:robot:interface:index"
          ...]
```

After the two mandatory parameters, the next most useful driver parameter is `model`. This is only used if `"stage"` is the driver, it tells Player which particular model in the worldfile is providing the interfaces for this particular driver. A different driver is needed for each model that you want to use. Models that aren't required to do anything (such as a map, or in the example of section 3.3 oranges and boxes) don't need to have a driver written for them.
The remaining driver parameters are `requires` and `plugin`. The `requires` is used for drivers that need input information such as `"vfh"`, it tells this driver where to find this information and which interface it uses. The `requires` parameter uses the same key:host:robot:interface:index syntax as the `provides` parameter. Finally the `plugin` parameter is used to tell Player where to find all the information about the driver being used. Earlier we made a .cfg file in order to create a simulation of an empty (or at least unmoving) world, the .cfg file read as follows:

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

      provides ["simulation:0" ]

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

This has to be done at the beginning of the configuration file because it tells Player that there is a driver called `"stage"` that we're going to use and the code for dealing with this driver can be found in the `stageplugin` plugin. This needs to be specified for Stage because Stage is an add-on for Player, for drivers that are built into Player by default the `plugin` doesn't need to be specified.

[3] http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group__interfaces.html

4.1 Device Addresses - key:host:robot:interface:index

A device address is used to tell Player where the driver you are making will present (or receive) information and which interface to use in order to read this information. This is a string in the form `key:host:robot:interface:index` where each field is separated by a colon.

- `key`: The Player manual states that: *"The purpose of the key field is to allow a driver that supports multiple interfaces of the same type to map those interfaces onto different devices"*[1]. This is a driver level thing and has a lot to do with the `name` of the driver that you are using, generally for `"stage"` the `key` doesn't need to be used. If you're using Player without Stage then there is a useful section about device address keys in the Player manual[4].

- `host`: This is the address of the host computer where the device is located. With a robot it could be the IP address of the robot. The default host is "localhost" which means the computer on which Player is running.

- `robot`: this is the TCP port through which Player should expect to receive data from the interface usually a single robot and all its necessary interfaces are assigned to one port. The default port used is 6665, if there were two robots in the simulation the ports could be 6665 and 6666 although there's no rule saying which number ports you can or can't use.

- `interface`: The interface to use in order to interact with the data. There is no default value for this option because it is a mandatory field.

- `index`: If a robot has multiple devices of the same type, for instance it has 2 cameras to give the robot depth perception, each device uses the same interface but gives slightly different information. The index field allows you to give a slightly different address to each device. So two cameras could be `camera:0` and `camera:1`. This is very different from the `key` field because having a "driver that supports multiple interfaces of the same type" is NOT the same as having multiple devices that use the same interface. Again there is no default index, as this is a mandatory field in the device address, but you should use 0 as the index if there is only one of that kind of device.

If you want to use any of the default values it can just be left out of the device address. So we could use the default host and robot port and specify (for example) a laser interface just by doing `"laser:0"`. However, if you want to specify fields at the beginning of the device address but not in the middle then the separating colons should remain. For example if we had a host at `"127.0.0.1"` with a `laser` interface then we would specify the address as `"127.0.0.1::laser:0"`, the robot field is empty but the colons around it are still there. You may notice that the key field here was left off as before.

[4] http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group__tutorial__config.html#device_key

4.2 Putting the Configuration File Together

We have examined the commands necessary to build a driver for a model in the worldfile, now it is just a case of putting them all together. To demonstrate this process we will build a configuration file for the worldfile developed in section 3. In this world we want our code to be able to interact with the robot, so in our configuration file we need to specify a driver for this robot.

```
driver
(
      # parameters...
)
```

The inbuilt driver that Player/Stage uses for simulations is called `"stage"` so the driver name is `"stage"`.

```
driver
(
      name "stage"
)
```

The Bigbob robot uses `position`, `laser`, `blobfinder` and `ranger` sensors. These correspond to the `position2d`, `laser`, `blobfinder` and interfaces respectively. The `ranger` sensor is a special case because the ranger interface has not currently been implemented (for at least version Stage 3.2.2 or earlier). To work around this you'll need to use the `sonar` interfaces instead (there is apparently an IR interface which could be used instead of `ranger`, but it doesn't seem to work on either Stage 2 or Stage 3).

We want our code to be able to read from these sensors, so we need to declare interfaces for them and tell Player where to find each device's data, for this we use the configuration file's `provides` parameter. This requires that we construct device addresses for each sensor; to remind ourselves, this is in the key:host:robot:interface:index format. We aren't using any fancy drivers, so we don't need to specify a key. We are running our robot in a simulation on the same computer as our Player sever, so the host name is `localhost` which is the default, so we also don't need to specify a host. The robot is a TCP port to receive robot information over, picking which port to use is pretty arbitrary but what usually happens is that the first robot uses the default port 6665 and subsequent robots use 6666, 6667, 6668 etc. There is only one robot in our simulation so we will use port 6665 for all our sensor information from this robot. We only have one sensor of each type, so our devices don't need separate indices. What would happen if we did have several sensors of the same type (like say two cameras) is that we put the first device at index 0 and subsequent devices using the same interface have index 1, then 2, then 3 and so on.[5] Finally we use interfaces appropriate to the sensors the robot has, so in our example these are the `position2d`, `laser`, `blobfinder` interfaces and for our `ranger` devices we will use `sonar`.

Putting all this together under the `provides` parameter gives us:

[5]There are lots of ranger sensors in our model but when we created the robot's sensors in section 3.2.2 we put them all into the same ranger model. So as far as the configuration file is concerned there is only one raging device using either the sonar or IR interface, because all the separate ranger devices are lumped together into this one model. We don't need to declare each ranger on an index of its own.

```
driver
(
  name "stage"
  provides ["6665:position2d:0"
            "6665:sonar:0"
            "6665:blobfinder:0"
            "6665:laser:0" ]
)
```

The device addresses can be on the same line as each other or separate lines, just so long as they're separated by some form of white space.

The last thing to do on our driver is the `model "model_name"` parameter which needs to be specified because we are using Player/Stage. This tells the simulation software that anything you do with this driver will affect the model `"model_name"` in the simulation. In the simulation we built we named our robot model "bob1", so our final driver for the robot will be:

```
driver
(
      name "stage"
      provides ["6665:position2d:0"
            "6665:sonar:0"
            "6665:blobfinder:0"
            "6665:laser:0"]
      model "bob1"
)
```

If our simulation had multiple Bigbob robots in, the configuration file drivers would be very similar to one another. If we created a second robot in our worldfile and called it "bob2" then the driver would be:

```
driver
(
      name "stage"
      provides ["6666:position2d:0"
            "6666:sonar:0"
            "6666:blobfinder:0"
            "6666:laser:0"]
      model "bob2"
)
```

Notice that the port number and model name are the only differences because the robots have all the same sensors.

A driver of this kind can be built for any model that is in the worldfile, not just the robots. For instance a map driver can be made which uses the `map` interface and will allow you to get size, origin and occupancy data about the map. The only requirement is that if you want to do something to the model with your code then you need to build a driver for it in the configuration file. Finally when we put the bit which declares the `stage` driver (this part is compulsory for any simulation configuration file) together with our drivers for the robot we end up with our final configuration file:

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

      provides ["simulation:0" ]

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

driver
(
      name "stage"
      provides ["6665:position2d:0"
            "6665:sonar:0"
            "6665:blobfinder:0"
            "6665:laser:0"]
      model "bob1"
)
```

Chapter 5

Getting Your Simulation To Run Your Code

To learn how to write code for Player or Player/Stage it helps to understand the basic structure of how Player works. Player uses a Server/Client structure in order to pass data and instructions between your code and the robot's hardware. Player is a server, and a hardware device[1] on the robot is subscribed as a client to the server via a thing called a *proxy*. The .cfg file associated with your robot (or your simulation) takes care of telling the Player server which devices are attached to it, so when we run the command `player some_cfg.cfg` this starts up the Player server and connects all the necessary hardware devices to the server. Figure 5.1 shows a basic block diagram of the structure of Player when implemented on a robot. In Player/Stage the same command will start the Player server and load up the worldfile in a simulation window, this runs on your computer and allows your code to interact with the simulation rather than hardware. Figure 5.2 shows a basic block diagram of the Player/Stage structure. Your code must also subscribe to the Player server so that it can access these proxies and hence control the robot. Player has functions and classes which will do all this for you, but you still need to actually call these functions with your code and know how to use them.

Player is compatable with C, C++ or Python code, however in this manual we will only really be using C++ because it is the simplest to understand. The process of writing Player code is mostly the same for each different language though. The Player and Player proxy functions have different names for each language, but work in more or less the same way, so even if you don't plan on using C++ or Stage this section will still contain helpful information.

Before beginning a project it is highly recommended that for any programs other than basic examples you should always wrap your Player commands around your own functions and classes so that all your code's interactions with Player are kept together the same file. This isn't a requirement of Player, it's just good practice. For example, if you upgrade Player or if for some reason your robot breaks and a certain function no longer works you only have to change part of a file instead of searching through all your code for places where Player

[1] remember, a device is a piece of hardware that uses a driver which conforms to an interface. See section 2.2

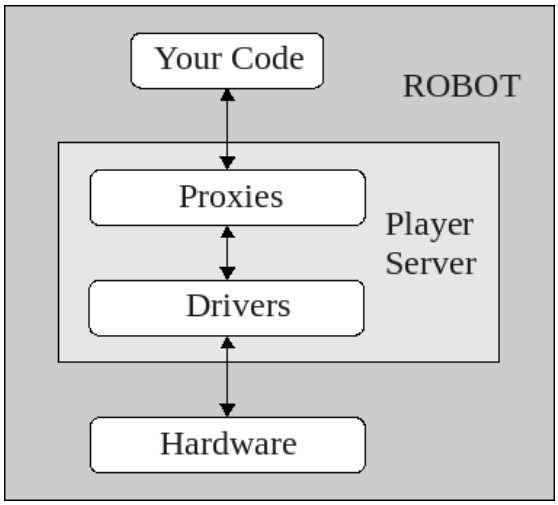

Figure 5.1: The server/client control structure of Player when used on a robot. There may be several proxies connected to the server at any time.

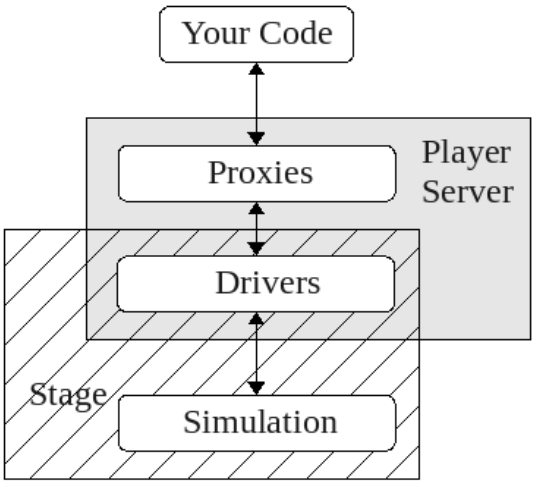

Figure 5.2: The server/client control structure of Player/Stage when used as a simulator. There may be several proxies connected to the server at any time.

functions have been used.

Finally, in order to compile your program you use the following commands (in Linux):

```
g++ -o example0 `pkg-config --cflags playerc++` example0.cc `pkg-config
--libs playerc++`
```

That will compile a program to a file called `example0` from the C++ code file `example0.cc`. If you are coding in C instead then use the following command:

```
gcc -o example0 `pkg-config --cflags playerc` example0.c `pkg-config
--libs playerc`
```

5.1 Connecting to the Server and Proxies With Your Code

The first thing to do within your code is to include the Player header file. Assuming Player/Stage is installed correctly on your machine then this can be done with the line `#include <libplayerc++/playerc++.h>` (if you're using C then type `#include <libplayerc/playerc.h>` instead).

Next we need to establish a Player Client, which will interact with the Player server for you. To do this we use the line:

```
PlayerClient client_name(hostname, port);
```

What this line does is declare a new object which is a PlayerClient called `client_name` which connects to the Player server at the given address. The hostname and port is like that discussed in section 4.1. If your code is running on the same computer (or robot) as the Player server you wish to connect to then the hostname is "localhost" otherwise it will be the IP address of the computer or robot. The port is an optional parameter usually only needed for simulations, it will be the same as the port you gave in the .cfg file. This is only useful if your simulation has more than one robot in and you need your code to connect to both robots. So if you gave your first robot port 6665 and the second one 6666 (like in the example of section 4.2) then you would need two PlayerClients, one connected to each robot, and you would do this with the following code:

```
PlayerClient robot1("localhost", 6665);
PlayerClient robot2("localhost", 6666);
```

If you are only using one robot and in your .cfg file you said that it would operate on port 6665 then the port parameter to the PlayerClient class is not needed.

Once we have established a PlayerClient we should connect our code to the device proxies so that we can exchange information with them. Which proxies you can connect your code to is dependent on what you have put in your configuration file. For instance if your configuration file says your robot is connected to a laser but not a camera you can connect to the laser device but not the camera, even if the robot (or robot simulation) has a camera on it.

Proxies take the name of the interface which the drivers use to talk to Player. Let's take part of the Bigbob example configuration file from section 4.2:

```
driver
(
  name "stage"
  provides ["6665:position2d:0"
            "6665:sonar:0"
            "6665:blobfinder:0"
            "6665:laser:0" ]
)
```

Here we've told the Player server that our "robot" has devices which use the position2d, sonar, blobfinder and laser interfaces. In our code then, we should connect to the position2d, sonar, blobfinder and laser proxies like so:

```
Position2dProxy positionProxy_name(&client_name,index);
SonarProxy      sonarProxy_name(&client_name,index);
BlobfinderProxy blobProxy_name(&client_name,index);
LaserProxy      laserProxy_name(&client_name,index);
```

A full list of which proxies Player supports can be found in the Player manual[2], they all follow the convention of being named after the interface they use. In the above case `xProxy_name` is the name you want to give to the proxy object, `client_name` is the name you gave the PlayerClient object earlier and `index` is the index that the device was given in your configuration file (probably 0).

5.1.1 Setting Up Connections: an Example.

For an example of how to connect to the Player sever and device proxies we will use the example configuration file developed in section 4.2. For convenience this is reproduced below:

```
driver
(
      name "stage"
      plugin "libstageplugin"

      provides ["simulation:0" ]

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

driver
(
      name "stage"
      provides ["6665:position2d:0"
            "6665:sonar:0"
            "6665:blobfinder:0"
            "6665:laser:0"]
      model "bob1"
)
```

[2]http://playerstage.sourceforge.net/doc/Player-2.1.0/player/classPlayerCc_1_1ClientProxy.html

To set up a PlayerClient and then connect to proxies on that server we can use principles discussed in this section to develop the following code:

```
#include <stdio.h>
#include <libplayerc++/playerc++.h>

int main(int argc, char *argv[])
{
      /*need to do this line in c++ only*/
      using namespace PlayerCc;

      PlayerClient    robot("localhost");

      Position2dProxy p2dProxy(&robot,0);
      SonarProxy      sonarProxy(&robot,0);
      BlobfinderProxy blobProxy(&robot,0);
      LaserProxy      laserProxy(&robot,0);

      //some control code
      return 0;
}
```

5.2 Interacting with Proxies

As you may expect, each proxy is specialised towards controlling the device it connects to. This means that each proxy will have different commands depending on what it controls. In Player version 2.1.0 there are 38 different proxies which you can choose to use, many of which are not applicable to Player/Stage. This manual will not attempt to explain them all, a full list of avaliable proxies and their functions is in the Player manual[3], although the returns, parameters and purpose of the proxy function is not always explained.
The following few proxies are probably the most useful to anyone using Player or Player/Stage.

5.2.1 Position2dProxy

The Position2dProxy is the number one most useful proxy there is. It controls the robot's motors and keeps track of the robot's odometry (where the robot thinks it is based on how far its wheels have moved).

Get/SetSpeed

The `SetSpeed` command is used to tell the robot's motors how fast to turn. There are two different `SetSpeed` commands that can be called, one is for robots that can move in any direction and the other is for robots with differential or car-like drives.

- `SetSpeed(double XSpeed, double YSpeed, double YawSpeed)`

[3] http://playerstage.sourceforge.net/doc/Player-2.1.0/player/classPlayerCc_1_1ClientProxy.html

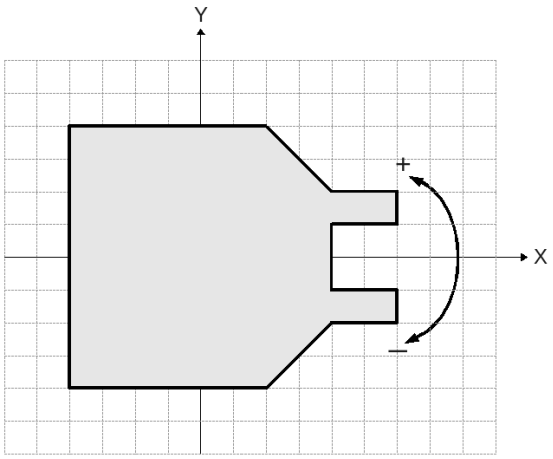

Figure 5.3: A robot on a cartesian grid. This shows what directions the X and Y speeds will cause the robot to move in. A positive yaw speed will turn the robot in the direction of the + arrow, a negative yaw speed is the direction of the - arrow.

- `SetSpeed(double XSpeed, double YawSpeed)`
- `SetCarlike(double XSpeed, double DriveAngle)`

Figure 5.3 shows which direction the x, y and yaw speeds are in relation to the robot. The x speed is the rate at which the robot moves forward and the y speed is the robot's speed sideways, both are to be given in metres per second. The y speed will only be useful if the robot you want to simulate or control is a ball, since robots with wheels cannot move sideways. The yaw speed controls how fast the robot is turning and is given in radians per second, Player has an inbuilt global function called `dtor()` which converts a number in degrees into a number in radians which could be useful when setting the yaw speed. If you want to simulate or control a robot with a differential drive system then you'll need to convert left and right wheel speeds into a forward speed and a turning speed before sending it to the proxy. For car-like drives there is the `SetCarlike` which, again is the forward speed in m/s and the drive angle in radians.

The `GetSpeed` commands are essentially the reverse of the SetSpeed command. Instead of setting a speed they return the current speed relative to the robot (so x is the forward speed, yaw is the turning speed and so on).

- `GetXSpeed`: forward speed (metres/sec).
- `GetYSpeed`: sideways (perpendicular) speed (metres/sec).
- `GetYawSpeed`: turning speed (radians/sec).

Get_Pos

This function interacts with the robot's odometry. It allows you to monitor where the robot thinks it is. Coordinate values are given relative to its starting point, and yaws are relative to its starting yaw.

- `GetXPos()`: gives current x coordinate relative to its x starting position.
- `GetYPos()`: gives current y coordinate relative to its y starting position.
- `GetYaw()`: gives current yaw relative to its starting yaw.

In section 3.2.1, we specified whether it would record odometry by measuring how much its wheels have turned, or whether the robot would have perfect knowledge of its current coordinates (by default the robot does not record odometry at all). If you set the robot to record odometry using its wheels then the positions returned by these get commands will become increasingly inaccurate as the simulation goes on. If you want to log your robots position as it moves around, these functions along with the perfect odometry[4] setting can be used.

SetMotorEnable()

This function takes a boolean input, telling Player whether to enable the motors or not. If the motors are disabled then the robot will not move no matter what commands are given to it, if the motors are enabled then the motors will always work, this is not so desirable if the robot is on a desk or something and is likely to get damaged. Hence the motors being enabled is optional. If you are using Player/Stage, then the motors will always be enabled and this command doesn't need to be run. However, if your code is ever likely to be moved onto a real robot and the motors are not explicitly enabled in your code, then you may end up spending a long time trying to work out why your robot is not working.

5.2.2 SonarProxy

The sonar proxy can be used to receive the distance from the sonar to an obstacle in metres. To do this you use the command:

- `sonarProxy_name[sonar_number]`

Where `sonarProxy_name` is the SonarProxy object and `sonar_number` is the number of the ranger. In Player/Stage the sonar numbers come from the order in which you described the ranger devices in the worldfile. In section 3.2.2 we described the ranger sensors for the Bigbob robot like so:

```
define bigbobs_sonars ranger
(
      # number of sonars
      scount 4

      # define the pose of each transducer [xpos ypos heading]
      spose[0] [ 0.75 0.1875 0 ] #fr left tooth
      spose[1] [ 0.75 -0.1875 0 ] #fr right tooth
      spose[2] [ 0.25 0.5 30] # left corner
      spose[3] [ 0.25 -0.5 -30] # right corner

)
```

[4]See section 3.2.1 for how to give the robot perfect odometry.

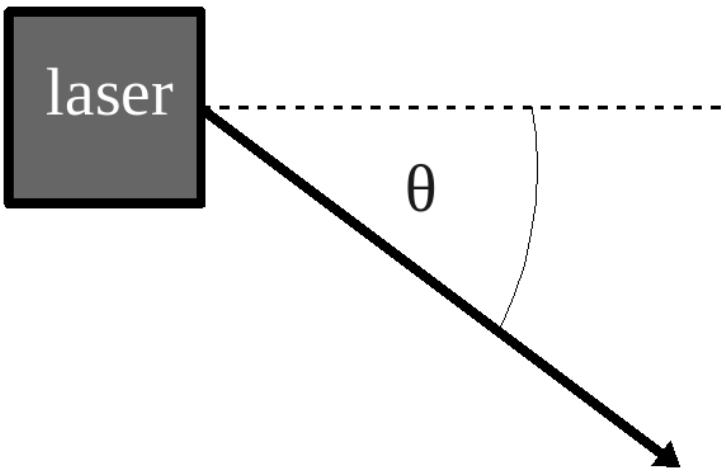

Figure 5.4: How laser angles are referenced. In this diagram the laser is pointing to the right along the dotted line, the angle θ is the angle of a laser scan point, in this example θ is negative.

In this example `sonarProxy_name[0]` gives us the distance from the left tooth ranger to an obstacle, `sonarProxy_name[1]` gives the distance from the front right tooth to an obstacle, and so on. If no obstacle is detected then the function will return whatever the ranger's maximum range is.

5.2.3 LaserProxy

A laser is a special case of ranger device, it makes regularly spaced range measurements turning from a minimum angle to a maximum angle. Each measurement, or scan point, is treated as being done with a separate ranger. Where angles are given they are given with reference to the laser's centre front (see figure 5.4).

- `GetCount`: The number of laser scan points that the laser measures.
- `laserProxy_name[laser_number]` The range returned by the `laser_number`th scan point. Scan points are numbered from the minimum angle at index 0, to the maximum angle at index `GetCount()`.
- `GetBearing[laser_number]`: This gets the angle of the laser scan point.
- `GetRange[laser_number]`: returns the range measured by the scan point `laser_number`. This is the same as doing `laserProxy_name[laser_number]`.
- `MinLeft`: Gives the minimum range returned by a scan point on the left hand side of the laser.
- `MinRight`: Gives the minimum range returned by a scan point on the right hand side of the laser.

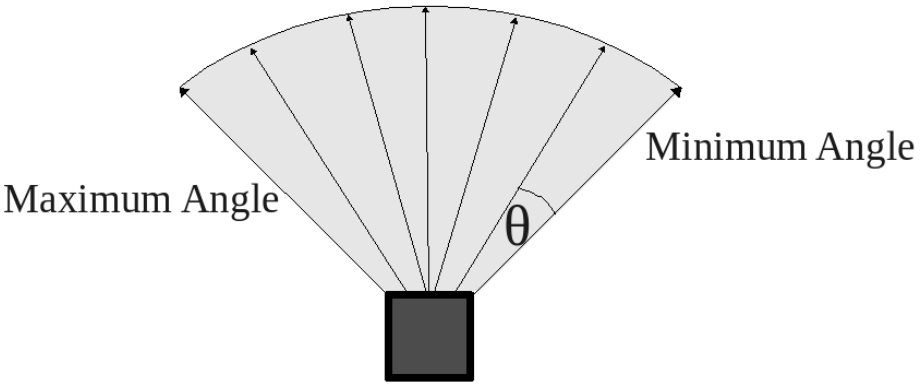

Figure 5.5: A laser scanner. The minimum angle is the angle of the rightmost laser scan, the maximum angle is the leftmost laser scan. θ is the scan resolution of the laser, it is the angle between each laser scan, given in 0.01 degrees.

5.2.4 RangerProxy

The RangerProxy is a proxy for more general ranging, it supports the sonar, laser and IR proxies. It has the same function as the SonarProxy in that you can use the code `rangerProxy_name[ranger_number]` to return the distance from ranger `ranger_number` to an obstacle. It also has many of the same functions as the LaserProxy, such as a minimum angle and a maximum angle and a scanning resolution, mostly these are for retrieving data about how the rangers are arranged or what size the ranging devices are.

5.2.5 BlobfinderProxy

The blobfinder module analyses a camera image for areas of a desired colour and returns an array of the structure `playerc_blobfinder_blob_t`, this is the structure used to store blob data. First we will cover how to get this data from the blobfinder proxy, then we will discuss the data stored in the structure.

- `GetCount`: Returns the number of blobs seen.
- `blobProxy_name[blob_number]`: This returns the blob structure data for the blob with the index `blob_number`. Blobs are sorted by index in the order that they appear in the image from left to right. This can also be achieved with the BlobfinderProxy function `GetBlob(blob_number)`.

Once we receive the blob structure from the proxy we can extract data we need. The `playerc_blobfinder_blob_t` structure contains the following fields:

- `color`: The colour of the blob it detected. This is given as a hexadecimal value.
- `area`: The area of the blob's bounding box.
- `x`: The horizontal coordinate of the geometric centre of the blob's bounding box (see figure 5.6).

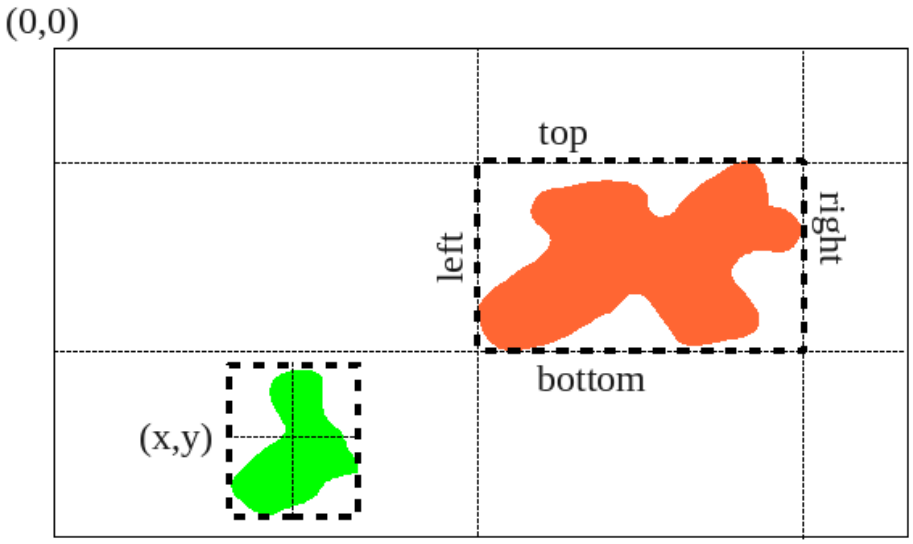

Figure 5.6: What the fields in playerc_blobfinder_blob_t mean. The blob on the left has a geometric centre at (x, y), the blob on the right has a bounding box with the top left corner at $(left, top)$ pixels, and a lower right coordinate at $(right, bottom)$ pixels. Coordinates are given with reference to the top left corner of the image.

- `y`: The vertical coordinate of the geometric centre of the blob's bounding box (see figure 5.6).
- `left`: The horizontal coordinate of the left hand side of the blob's bounding box (see figure 5.6).
- `right`: The horizontal coordinate of the right hand side of the blob's bounding box (see figure 5.6).
- `top`: The vertical coordinate of the top side of the blob's bounding box (see figure 5.6).
- `bottom`: The vertical coordinate of the bottom side of the blob's bounding box (see figure 5.6).

5.2.6 GripperProxy

The GripperProxy allows you to control the gripper, once the gripper is holding an item, the simulated robot will carry it around wherever it goes. Without a gripper you can only jostle an item in the simulation and you would have to manually tell the simulation what to do with an item. The GripperProxy can also tell you if an item is between the gripper teeth because the gripper model has inbuilt beams which can detect if they are broken.

- `GetBeams`: This command will tell you if there is an item inside the gripper. If it is a value above 0 then there is an item to grab.

- `GetState`: This will tell you whether the gripper is opened or closed. If the command returns a 1 then the gripper is open, if it returns 2 then the gripper is closed.
- `Open`: Tells the gripper to open. This will cause any items that were being carried to be dropped.
- `Close`: Tells the gripper to close. This will cause it to pick up anything between its teeth.

5.2.7 SimulationProxy

The simulation proxy allows your code to interact with and change aspects of the simulation, such as an item's pose or its colour.

Get/Set Property

To change a property of an item in the simulation we use the following function:
`SetProperty(char *item_name, char *property, void *value, size_t value_len)`

- `item_name`: this is the name that you gave to the object in the worldfile, it could be *any* model that you have described in the worldfile. For example, in section 3.2.2 in the worldfile we declared a Bigbob type robot which we called "bob1" so the `item_name` for that object is "bob1". Similarly in section 3.3 we built some models of oranges and called the "orange1" to "orange4" so the item name for one of these would be "orange1". Anything that is a model in your worldfile can be altered by this function, you just need to have named it, no drivers need to be declared in the configuration file for this to work either. We didn't write drivers for the oranges but we could still alter their properties this way.
- `property`: There are only certain properties about a model that you can change. You specify which one you want with a string. Since the properties you can change are fixed then these strings are predefined (see stage.hh[5]):
  - `"_mp_color"`: The colour of the item.
  - `"_mp_watts"`: The number of watts the item needs.
  - `"_mp_mass"`: The mass of the item.
  - `"_mp_fiducial_return"`: sets whether the item is detectable to the fiducial finder on a robot.
  - `"_mp_laser_return"`: sets whether the item is detectable to the laser on a robot.
  - `"_mp_obstacle_return"`: sets whether the robot can collide with the item.
  - `"_mp_ranger_return"`: sets whether the item is detectable to the rangers on a robot.
  - `"_mp_gripper_return"`: sets whether the item can be gripped by a gripper or jostled by the robot colliding with it.

[5] `http://playerstage.sourceforge.net/doc/Stage-3.2.1/stage_8hh_source.html`

- `value`: The value you want to assign to the property. For the return parameters this can simply be a 0 or a 1, for the watts and mass it can be a numerical value. For the colour of the item this is a `uint32_t` 8 digit long hexadecimal number. The first 2 digits are the alpha value of the colour, the second two are its red value, the next two are its green value and the final two are the blue colour value. So red, for instance, would be uint32_t red = 0xffff0000, green would be 0xff00ff00 and blue is 0xff0000ff. A nice shade of yellow might be 0xffffcc11.

- `value_len`: is the size of the value you gave in bytes. This can easily be found with the C or C++ `sizeof()` operator.

Similarly the following function can be used to get property information:
`GetProperty(char *item_name, char *property, void *value, size_t value_len)`
Instead of setting the given property of the item, this will write it to the memory block pointed to by `*value`.

Get/Set Pose

The item's pose is a special case of the Get/SetProperty function, because it is likely that someone would want to move an item in the world they created a special function to do it.
`SetPose2d(char *item_name, double x, double y, double yaw)`
In this case `item_name` is as with Get/SetProperty, but we can directly specify its new coordinates and yaw (coordinates and yaws are given with reference to the map's origin).
`GetPose2d(char *item_name, double &x, double &y, double &yaw)`
This is like SetPose2d only this time it writes the coordinates and yaw to the given addresses in memory.

5.2.8 General Useful Commands

Read()

To make the proxies update with new sensor data we need to tell the player server to update, we can do this using the PlayerClient object which we used to connect to the server. All we have to do is run the command `playerClient_name.Read()` every time the data needs updating (where playerClient_name is the name you gave the PlayerClient object). Until this command is run, the proxies and any sensor information from them will be empty. The devices on a typical robot are asynchronous and the devices in a Player/Stage simulation are also asynchronous, so running the `Read()` command won't always update everything at the same time, so it may take several calls before some large data structures (such as a camera image) gets updated.

GetGeom()

Most of the proxies have a function called `GetGeom` or `GetGeometry` or `RequestGeometry`, or words to that effect. What these functions do is tell the proxy retrieve information about the device, usually its size and pose (relative to the robot). The proxies don't know this by default since this information is specific to the

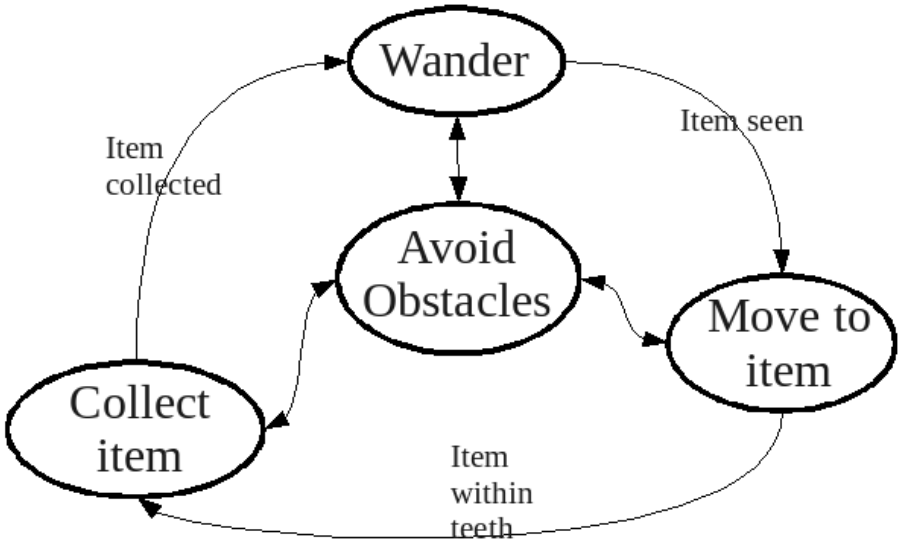

Figure 5.7: The state transitions that the Bigbob rubbish collecting robot will follow.

robot or the Player/Stage robot model. If your code needs to know this kind of information about a device then the proxy must run this command first.

5.3 Using Proxies: A Case Study

To demonstrate how to write code to control a Player device or Player/Stage simulation we will use the example robot "Bigbob" developed in sections 3.2 and 4 which collects oranges and juice cartons from a factory floor. In previous sections we have developed the Stage model for this robot and its environment and the configuration file to control it. Now we can begin to put everything together to create a working simulation of this robot.

5.3.1 The Control Architecture

To collect rubbish we have three basic behaviours:

- Wandering: to search for rubbish.
- Moving towards item: for when an item is spotted and the robot wants to collect it
- Collecting item: for dealing with collecting items.

The robot will also avoid obstacles but once this is done it will switch back to its previous behaviour. The control will follow the state transitions shown in figure 5.7.

5.3.2 Beginning the Code

In section 5.1.1 we discussed how to connect to the Player server and proxies attached to the server, and developed the following code:

```
#include <stdio.h>
#include <libplayerc++/playerc++.h>

int main(int argc, char *argv[])
{
      /*need to do this line in c++ only*/
      using namespace PlayerCc;

      PlayerClient    robot("localhost");

      Position2dProxy p2dProxy(&robot,0);
      SonarProxy      sonarProxy(&robot,0);
      BlobfinderProxy blobProxy(&robot,0);
      LaserProxy      laserProxy(&robot,0);

      //some control code
      return 0;
}
```

Using our knowledge of the proxies discussed in section 5.2 we can build controlling code on top of this basic code. Firstly, it is good practice to enable the motors and request the geometry for all the proxies. This means that the robot will move and that if we need to know about the sensing devices the proxies will have that information available.

```
//enable motors
p2dProxy.SetMotorEnable(1);

//request geometries
p2dProxy.RequestGeom();
sonarProxy.RequestGeom();
laserProxy.RequestGeom();
//blobfinder doesn't have geometry
```

Once things are initialised we can enter the main control loop. At this point we should tell the robot to read in data from its devices to the proxies.

```
while(true)
{
      robot.Read();

      //control code
}
```

5.3.3 Wander

first we will initialise a couple of variables which will be the forward speed and the turning speed of the robot, we'll put this with the proxy initialisations.

```
Position2dProxy p2dProxy(&robot,0);
SonarProxy      sonarProxy(&robot,0);
BlobfinderProxy blobProxy(&robot,0);
LaserProxy      laserProxy(&robot,0);

double forwardSpeed, turnSpeed;
```

Let's say that Bigbob's maximum speed is 1 metre/second and it can turn 90° a second. We will write a small subfunction to randomly assign forward and turning speeds between 0 and the maximum speeds.

```
void Wander(double *forwardSpeed, double *turnSpeed)
{
      int maxSpeed = 1;
      int maxTurn = 90;
      double fspeed, tspeed;

      //fspeed is between 0 and 10
      fspeed = rand()%11;
      //(fspeed/10) is between 0 and 1
      fspeed = (fspeed/10)*maxSpeed;

      tspeed = rand()%(2*maxTurn);
      tspeed = tspeed-maxTurn;
      //tspeed is between -maxTurn and +maxTurn

      *forwardSpeed = fspeed;
      *turnSpeed = tspeed;
}
```

In the control loop we include a call to this function and then set the resulting speeds to the motors.

```
while(true)
{
      // read from the proxies
      robot.Read();

      //wander
      Wander(&forwardSpeed, &turnSpeed);

      //set motors
      p2dProxy.SetSpeed(forwardSpeed, dtor(turnSpeed));
}
```

At present the motors are being updated every time this control loop executes, and this leads to some erratic behaviour from the robot. Using the `sleep()`[6] command we will tell the control loop to wait one second between each execution. At this point we should also seed the random number generator with the current time so that the wander behaviour isn't exactly the same each

[6]sleep() is a standard C function and is included in the unistd.h header.

time. For the sleep command we will need to include `unistd.h` and to seed the random number generator with the current system time we will need to include `time.h`.

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <libplayerc++/playerc++.h>

void Wander(double *forwardSpeed, double *turnSpeed)
{
      //wander code...
}

int main(int argc, char *argv[])
{
      /*need to do this line in c++ only*/
      using namespace PlayerCc;

      //connect to proxies
      double forwardSpeed, turnSpeed;

      srand(time(NULL));

      //enable motors
      //request geometries

      while(true)
      {
            // read from the proxies
            robot.Read();

            //wander
            Wander(&forwardSpeed, &turnSpeed);

            //set motors
            p2dProxy.SetSpeed(forwardSpeed, dtor(turnSpeed));
            sleep(1);
      }
}
```

5.3.4 Obstacle Avoidance

Now we need to write a subfunction that checks the sonars for any obstacles and amends the motor speeds accordingly.

```
void AvoidObstacles(double *forwardSpeed, double *turnSpeed, \
      SonarProxy &sp)
{
      //will avoid obstacles closer than 40cm
      double avoidDistance = 0.4;
```

```
      //will turn away at 60 degrees/sec
      int avoidTurnSpeed = 60;

      //left corner is sonar no. 2
      //right corner is sonar no. 3
      if(sp[2] < avoidDistance)
      {
            *forwardSpeed = 0;
            //turn right
            *turnSpeed = (-1)*avoidTurnSpeed;
            return;
      }
      else if(sp[3] < avoidDistance)
      {
            *forwardSpeed = 0;
            //turn left
            *turnSpeed = avoidTurnSpeed;
            return;
      }
      else if( (sp[0] < avoidDistance) && \
               (sp[1] < avoidDistance))
      {
            //back off a little bit
            *forwardSpeed = -0.2;
            *turnSpeed = avoidTurnSpeed;
            return;
      }

      return; //do nothing
}
```

This is a very basic obstacle avoidance subfunction will update the motor speeds only if there is an obstacle to avoid. If we call this function just before sending data to the motors then it will overwrite any other behaviours so that the obstacle will be avoided. Once the obstacle is no longer in the way then the robot will continue as it was, this will allow us to transition from any behaviour into obstacle avoidance and then back again, as per the requirement of our control structure. All we need to do now is call this function in our control loop:

```
while(true)
{
    // read from the proxies
    robot.Read();

    //wander
    Wander(&forwardSpeed, &turnSpeed);

    //avoid obstacles
    AvoidObstacles(&forwardSpeed, &turnSpeed, sonarProxy);
```

```
    //set motors
    p2dProxy.SetSpeed(forwardSpeed, dtor(turnSpeed));
    sleep(1);
}
```

5.3.5 Move To Item

For this state we want the robot to move towards a blob that it has spotted. There may be several blobs in its view at once, so we'll tell the robot to move to the largest one because it's probably the closest to the robot. The following subfunction finds the largest blob and turns the robot so that the blob's centre is near the centre of the image. The robot will then move towards the blob.

```
void MoveToItem(double *forwardSpeed, double *turnSpeed, \
      BlobfinderProxy &bfp)
{
      int i, centre;
      int noBlobs = bfp.GetCount();
      playerc_blobfinder_blob_t blob;
      int turningSpeed = 10;

      /*number of pixels away from the image centre a blob
      can be to be in front of the robot*/
      int margin = 10;

      int biggestBlobArea = 0;
      int biggestBlob = 0;

      //find the largest blob
      for(i=0; i<noBlobs; i++)
      {
            //get blob from proxy
            playerc_blobfinder_blob_t currBlob = bfp[i];

            if(currBlob.area > biggestBlobArea)
            {
                  biggestBlob = i;
                  biggestBlobArea = currBlob.area;
            }
      }
      blob = bfp[biggestBlob];

      // find centre of image
      centre = bfp.GetWidth()/2;

      //adjust turn to centre the blob in image
      /*if the blob's centre is within some margin of the image
      centre then move forwards, otherwise turn so that it is
      centred. */
      //blob to the left of centre
```

```
	if(blob.x < centre-margin)
	{
		*forwardSpeed = 0;
		//turn left
		*turnSpeed = turningSpeed;
	}
	//blob to the right of centre
	else if(blob.x > centre+margin)
	{
		*forwardSpeed = 0;
		//turn right
		*turnSpeed = -turningSpeed;
	}
	//otherwise go straight ahead
	else
	{
		*forwardSpeed = 0.5;
		*turnSpeed = 0;
	}

	return;
}
```

We want the robot to transition to this state whenever an item is seen, so we put a conditional statement in our control loop like so:

```
if(blobProxy.GetCount() == 0)
{
	//wander
	Wander(&forwardSpeed, &turnSpeed);
}
else
{
	//move towards the item
	MoveToItem(&forwardSpeed, &turnSpeed, blobProxy);
}
```

5.3.6 Collect Item

This behaviour will be the most difficult to code because Stage doesn't support pushable objects (the required physics is far too complex), what happens instead is that the robot runs over the object and just jostles it a bit. As a work-around to this problem we will have to somehow find out which item is between Bigbob's teeth so that we can find its "name" and then change that item's pose (for which we need the item's name) so that it is no longer in the simulation. In essence, instead of having our robot eat rubbish and store it within its body, what we are doing is making the laser zap the rubbish out of existence.

We can find the name of an item between Bigbob's teeth by cross referencing the robot's pose with the locations of the items in the world to find out which item is nearest the robot's laser. The first step is to create a list of all the items in the world, their names and their poses at initialisation. Since we

know the names of the items are "orange1" to "orange4" and "carton1" to "carton4", we can find their poses with a simple call to a simulation proxy. We'll have to connect to the simulation proxy with our code first using the line `SimulationProxy simProxy(&robot,0);`, then we can access this information and put it into a struct.

```
struct Item
{
      char name[16];
      double x;
      double y;
}typedef item_t;
```

We can populate the structure with information using the following code:

```
item_t itemList[8];

void RefreshItemList(item_t *itemList, SimulationProxy &simProxy)
{
      int i;

      //get the poses of the oranges
      for(i=0;i<4;i++)
      {
            char orangeStr[] = "orange%d";
            sprintf(itemList[i].name, orangeStr, i+1);
            double dummy;  //dummy variable, don't need yaws.
            simProxy.GetPose2d(itemList[i].name, \
                  itemList[i].x, itemList[i].y, dummy);
      }

      //get the poses of the cartons
      for(i=4;i<8;i++)
      {
            char cartonStr[] = "carton%d";
            sprintf(itemList[i].name, cartonStr, i-3);
            double dummy;  //dummy variable, don't need yaws.
            simProxy.GetPose2d(itemList[i].name, \
                  itemList[i].x, itemList[i].y, dummy);
      }

      return;
}
```

Where `itemList` is an `item_t` array of length 8.

Next we can begin the "Collect Item" behaviour, which will be triggered by something breaking the laser beam. When this happens we will check the area around Bigbob's teeth, as indicated by figure 5.8. We know the distance from the centre of this search circle to Bigbob's origin (0.625m) and the radius of the search circle (0.375m), we can get the robot's exact pose with the following code.

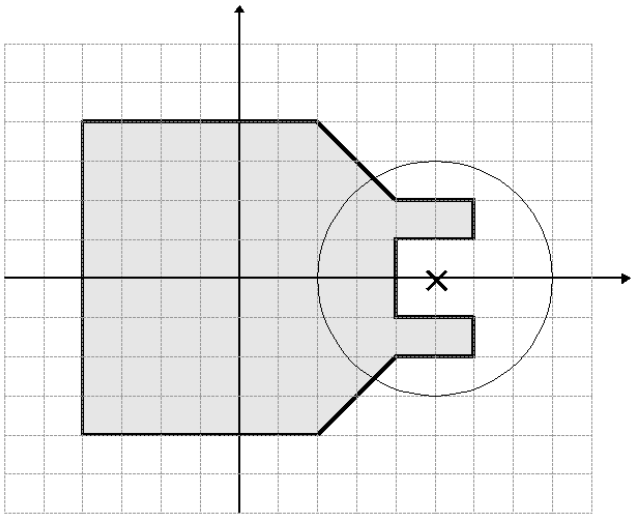

Figure 5.8: Where to look for items which may have passed through Bigbob's laser.

```
double x, y, yaw;
simProxy.GetPose2d("bob1", x, y, yaw);
```

Cross referencing the robot's position with the item positions is a matter of trigonometry, we won't reproduce the code here, but the full and final code developed for the Bigbob rubbish collecting robot is included in appendix D. The method we used is to find the Euclidian distance of the items to the circle centre, and the smallest distance is the item we want to destroy. We made a subfunction called `FindItem` that returns the index of the item to be destroyed.

Now that we can find the item to destroy it's fairly simple to trigger our subfunction when the laser is broken so we can find and destroy an item.

```
if(laserProxy[90] < 0.25)
{
      int destroyThis;

      /*first param is the list of items in the world
      second is length of this list
      third parameter is the simulation proxy with
      the pose information in it*/
      destroyThis = FindItem(itemList, 8, simProxy);

      //move it out of the simulation
      simProxy.SetPose2d(itemList[destroyThis].name, -10, -10, 0);
      RefreshItemList(itemList, simProxy);
}
```

The laser has 180 points, so point number 90 is the one which is perpendicular to Bigbob's teeth. This point returns a maximum of 0.25, so if its range was to fall below this then something has passed through the laser beam. We then

find the item closest to the robot's teeth and move that item to coordinate $(-10, -10)$ so it is no longer visible or accessible.
Finally we have a working simulation of a rubbish collecting robot! The full code listing is included in appendix D, the simulation world and configuration files are in appendices B and C respectively.

5.4 Simulating Multiple Robots

Our robot simulation case study only shows how to simulate a single robot in a Player/Stage environment. It's highly likely that a simulation might want more than one robot in it. In this situation you will need to build a model of every robot you need in the worldfile, and then its associated driver in the configuration file. Let's take a look at our worldfile for the case study, we'll add a new model of a new Bigbob robot called "bob2":

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

bigbob
(
name "bob2"
pose [5 6 225]
color "yellow"
)
```

If there are multiple robots in the simulation, the standard practice is to put each robot on its own port (see section 4.1). To implement this in the configuration file we need to tell Player which port to find our second robot on:

```
driver( name "stage"
        provides ["6665:position2d:0" "6665:sonar:0"
        "6665:blobfinder:0" "6665:laser:0"]
        model "bob1" )

driver( name "stage"
        provides ["6666:position2d:0" "6666:sonar:0"
        "6666:blobfinder:0" "6666:laser:0"]
        model "bob2" )
```

If you plan on simulating a large number of robots then it is probably worth writing a script to generate the world and configuration files.

When Player/Stage is started, the Player server automatically connects to all the used ports in your simulation and you control the robots separately with different PlayerClient objects in your code. For instance:

```
//first robot
PlayerClient robot1("localhost", 6665);
```

```
Position2dProxy p2dprox1(&robot1,0);
SonarProxy sprox1(&robot1,0);

//second robot
PlayerClient robot2("localhost", 6666);
Position2dProxy p2dprox2(&robot2,0);
SonarProxy sprox2(&robot2,0);
```

Each Player Client represents a robot, this is why when you connect to a proxy the PlayerClient is a constructor parameter. Each robot has a proxy for each of its devices, no robots share a proxy, so it is important that your code connects to every proxy of every robot in order to read the sensor information. How you handle the extra PlayerClients and proxies is dependent on the scale of the simulation and your own personal coding preferences. It's a good idea, if there's more than maybe 2 robots in the simulation, to make a robot class which deals with connecting to proxies and the server, and processes all the information internally to control the robot. Then you can create an instance of this class for each simulated robot[7] and all the simulated robots will run the same code.

An alternative to using a port for each robot is to use the same port but a different index. This will only work if the robots are all the same (or at least use the same interfaces, although different robots could be run on a different ports) and the robots only use one index for each of its devices. For example, the Bigbob robot uses interfaces and indexes: position2d:0, sonar:0, blobfinder:0 and laser:0 so it never uses more than one index. If we configured two Bigbob robots to use the same port but a different index our configuration file would be like this:

```
driver( name "stage"
        provides ["6665:position2d:0" "6665:sonar:0"
        "6665:blobfinder:0" "6665:laser:0"]
        model "bob1" )

driver( name "stage"
        provides ["6665:position2d:1" "6665:sonar:1"
        "6665:blobfinder:1" "6665:laser:1"]
        model "bob2" )
```

In our code we could then establish the proxies using only one PlayerClient:

```
PlayerClient robot("localhost", 6665);

//first robot
Position2dProxy p2dprox1(&robot,0);
SonarProxy sprox1(&robot,0);

//second robot
Position2dProxy p2dprox2(&robot,1);
SonarProxy sprox2(&robot,1);
```

[7] obviously the robot's port number would need to be a parameter otherwise they'll all connect to the same port and consequently the same robot.

```
//shared Simultion proxy...
SimulationProxy sim(&robot,0);
```

The main advantage of configuring the robot swarm this way is that it allows us to only have one simulation proxy which is shared by all robots. This is good since there is only ever one simulation window that you can interact with and so multiple simulation proxies are unnecessary.

Chapter 6

Useful Links

- Player 2.1.0 Manual
  http://playerstage.sourceforge.net/doc/Player-2.1.0/player/
- Stage 3.0.1 Manual
  http://playerstage.sourceforge.net/doc/stage-3.0.1/
- Stage 2.0.0 Manual
  http://playerstage.sourceforge.net/doc/Stage-2.0.0/
- All Player Proxies in C
  http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group_
  _playerc__proxies.html
- All Player Proxies in C++
  http://playerstage.sourceforge.net/doc/Player-2.1.0/player/namespacePlayerCc.
  html
- Interfaces used by Player
  http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group_
  _interfaces.html

Bibliography

[1] Brian Gerkey, Richard Vaughan, and Andrew Howard. Player manual: Device addresses. http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group__tutorial__config.html#device_addresses.