How to Use Player/Stage

Jennifer Owen

April 14, 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as there are significant differences with the previous versions of Stage and the previous edition of this manual is now out of date.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

1 Introduction **3**
1.1 A Note on Installing Player/Stage 3

2 The Basics **5**
2.1 Important File Types 5
2.2 Interfaces, Drivers and Devices 6

3 Building a World **8**
3.1 Building an Empty World 8
3.1.1 Models 10
3.1.2 Describing the Player/Stage Window 13
3.1.3 Making a Basic Worldfile 14
3.2 Building a Robot 15
3.2.1 Sensors and Devices 15
3.2.2 An Example Robot 19

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 **??**.

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 **??**, 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

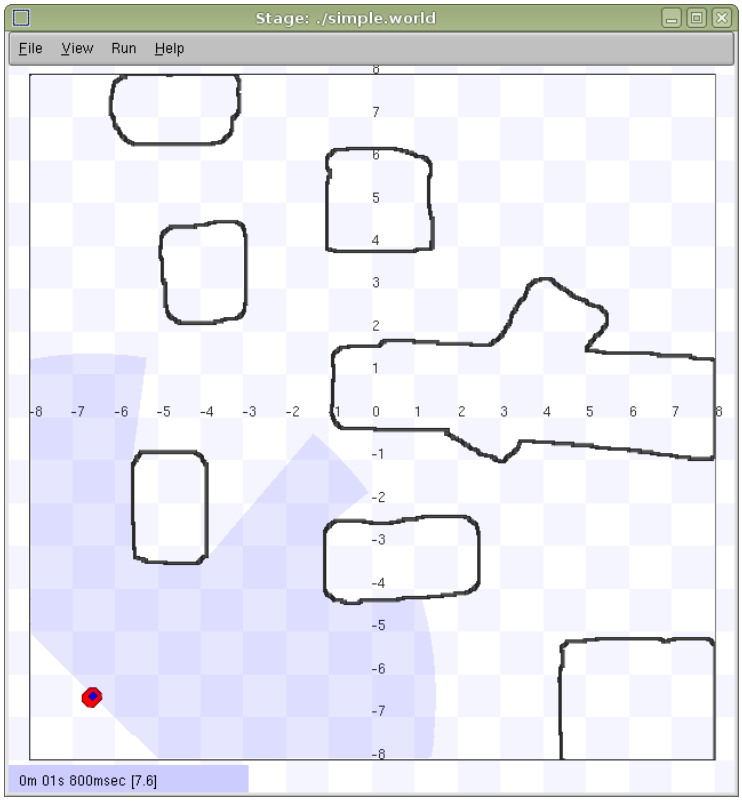

Figure 3.1: The simple.cfg world after being run

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

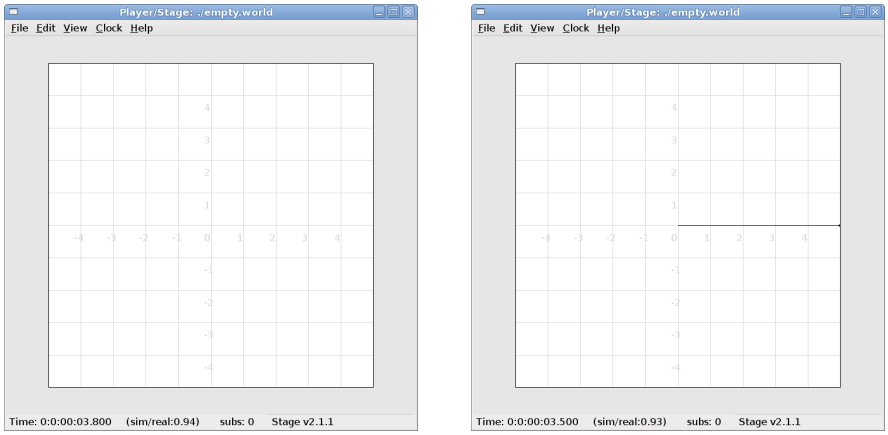

Figure 3.2: The left picture shows an empty map without a nose. The right picture shows the same map with a nose to indicate orientation, this is the horizontal line from the centre of the map to the right, it shows that the map is actually facing to the right.

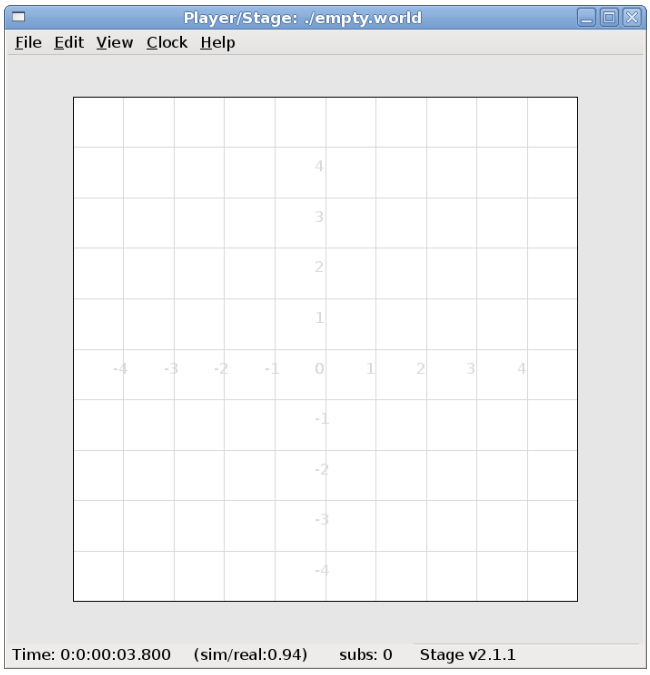

Figure 3.3: An empty map with gui_grid enabled. With gui_grid disabled this would just be an empty white square.

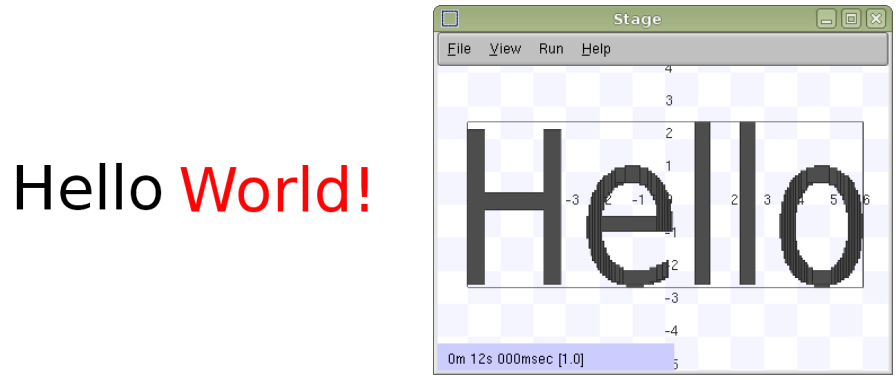

Figure 3.4: The left image is our "helloworld.png" bitmap, the right image is what Player/Stage interprets that bitmap as. The coloured areas are walls, the robot can move everywhere else.

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

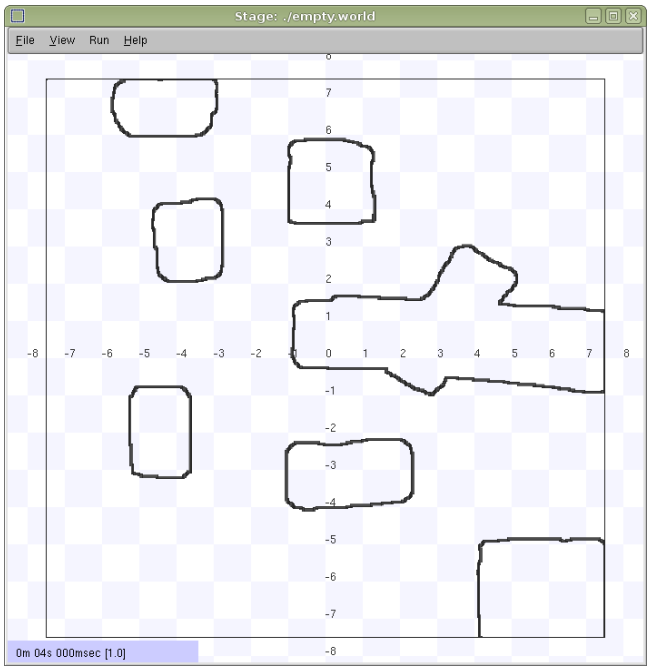

Figure 3.5: Our Empty World.

```
# 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 **??**. 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

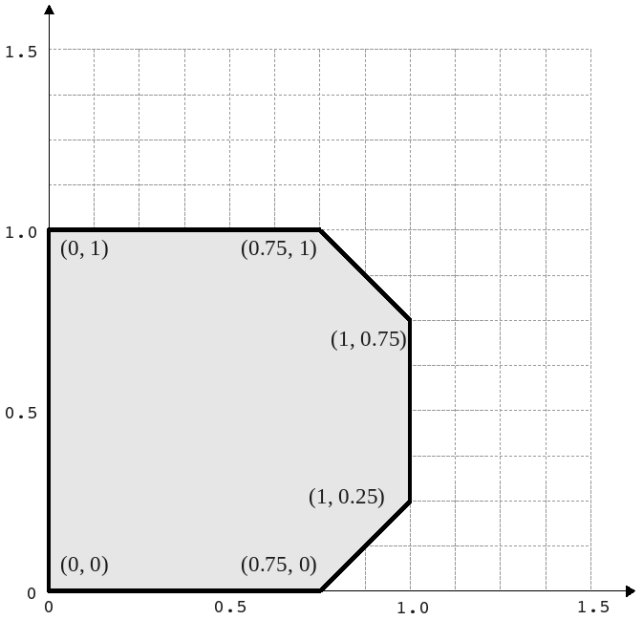

Figure 3.6: The basic shape we want to make Bigbob, the units on the axes are in metres.

- `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.

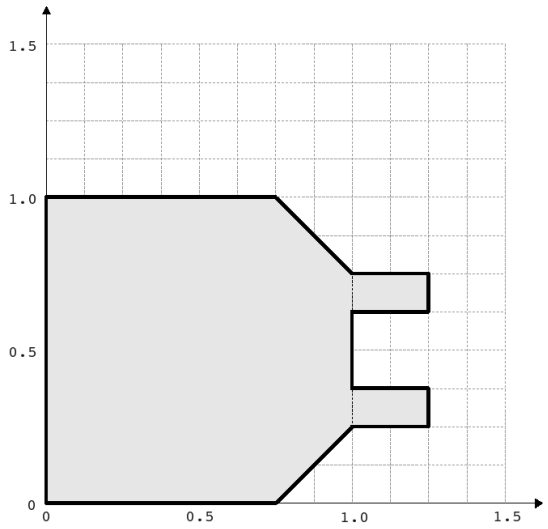

Figure 3.7: The new shape of Bigbob.

```
            point[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]
      )
)
```

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

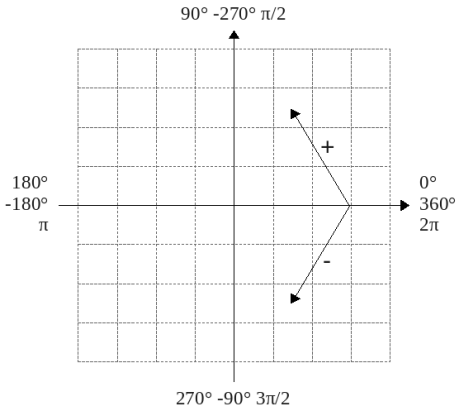

Figure 3.8: A cartesian grid showing how angles are described.

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
```

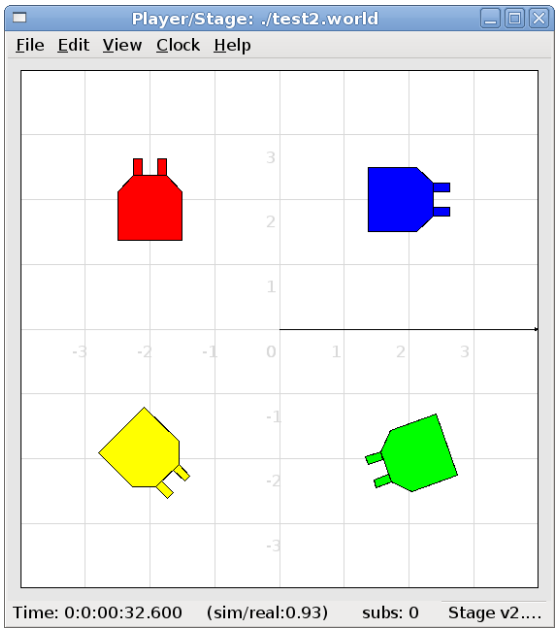

Figure 3.9: Starting from the top right robot and working anti-clockwise, the yaws of these robots are 0, 90, -45 and 200.

```
(
      # 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:

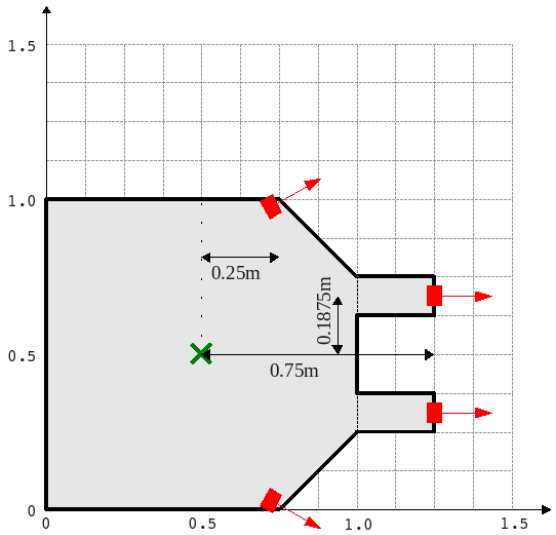

Figure 3.10: The position of Bigbob's sonars (in red) relative to its origin. The origin is marked with a cross, some of the distances from the origin to the sensors have been marked. The remaining distances can be done by inspection.

```
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
)
```