

How to Use Player/Stage

Jennifer Owen

March 17, 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 community 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as there are significant differences with the previous versions of Stage and the previous edition of this manual is now out of date.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

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 Player¹, 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.

¹Or 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 manual². 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

²http://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 ??.

3.1 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

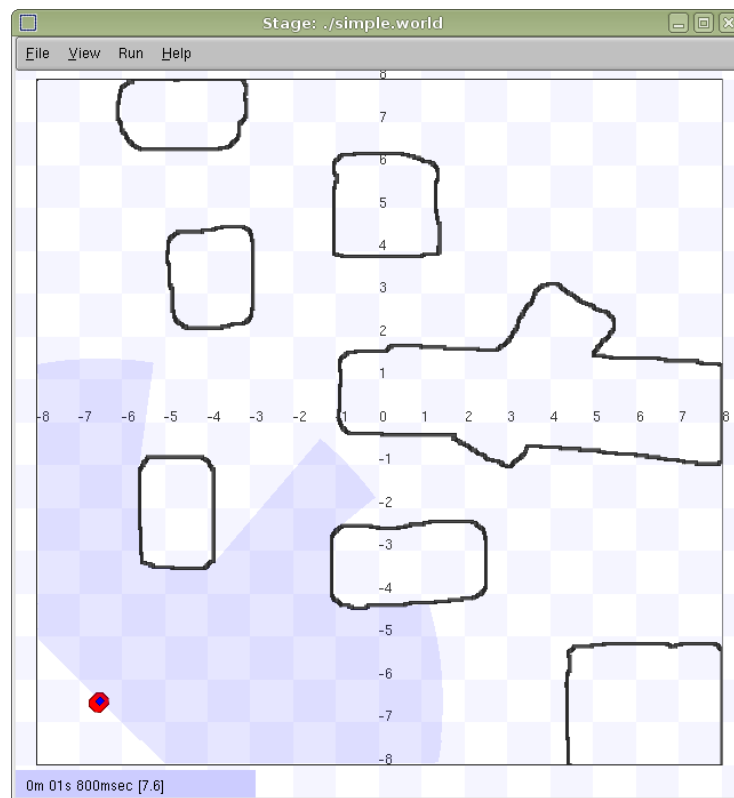


Figure 3.1: The simple.cfg world after being run

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 ?? . 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

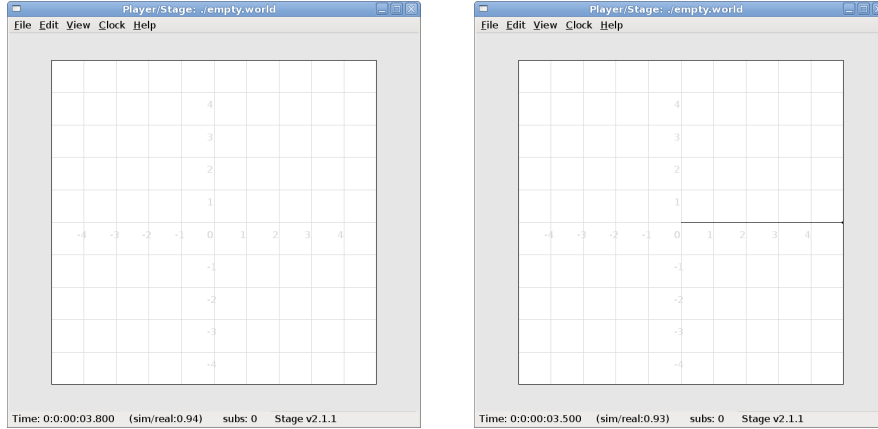


Figure 3.2: The left picture shows an empty map without a nose. The right picture shows the same map with a nose to indicate orientation, this is the horizontal line from the centre of the map to the right, it shows that the map is actually facing to the right.

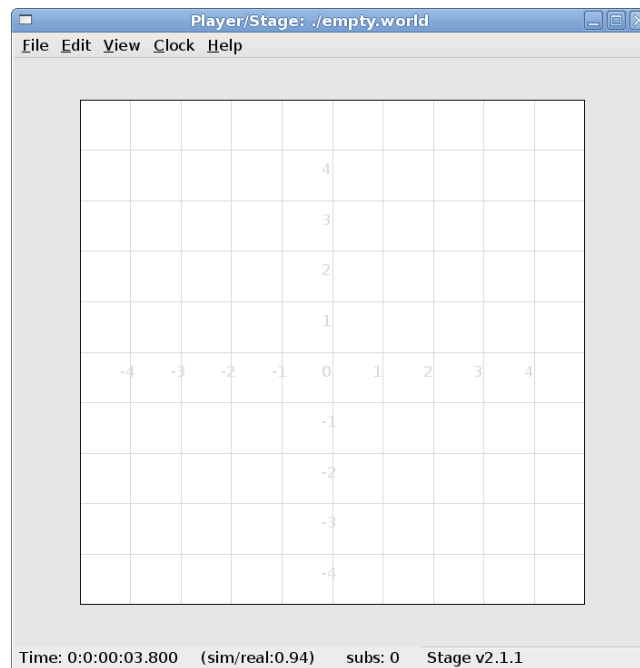


Figure 3.3: An empty map with gui_grid enabled. With gui_grid disabled this would just be an empty white square.

Hello World!



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 manual¹. 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 ??.

3.1.2 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:

¹http://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 $\frac{\text{window_size}}{\text{floorplan_size}}$ 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 parameters³. Finally, we are able to write a worldfile!

```

include "map.inc"

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

```

²http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__worldgui.html

³http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__world.html

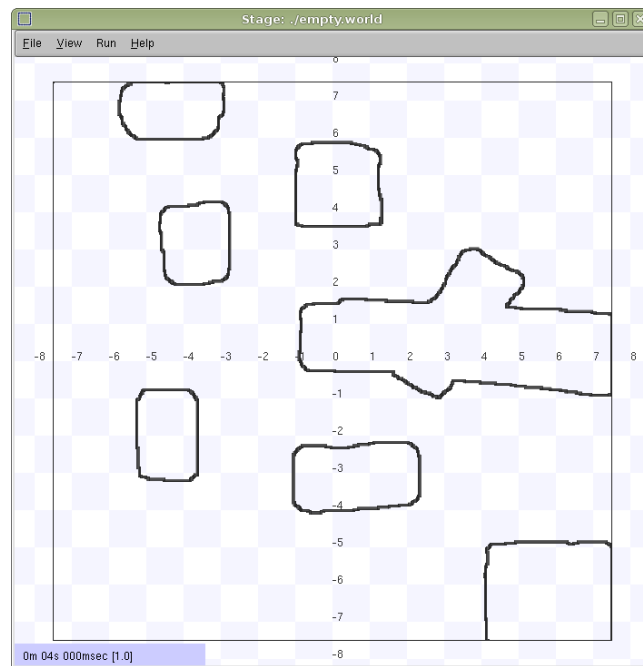


Figure 3.5: Our Empty World.

```
# 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.